2nd International Workshop on
FPGAs for Software Programmers (FSP 2015)

London, United Kingdom, September 1, 2015

co-located with
International Conference on Field Programmable Logic and Applications (FPL)

Edited by Frank Hannig, Dirk Koch, and Daniel Ziener

ePrint Proceedings: arXiv:1508.06320 [cs.AR]

Proceedings of the Second International Workshop on

FPGAs for Software Programmers (FSP 2015)

London, United Kingdom, September 1, 2015

co-located with
International Conference on Field Programmable Logic and Applications (FPL)

Edited by
Frank Hannig, Dirk Koch, and Daniel Ziener

Table of Contents

2nd International Workshop on FPGAs for Software Programmers (FSP 2015)

Message from the Workshop Chairs

This volume contains the papers accepted at the Second International Workshop on FPGAs for Software Programmers (FSP 2015), held in London, United Kingdom, September 1st, 2015. FSP 2015 was co-located with the International Conference on Field Programmable Logic and Applications (FPL).

The aim of the FSP workshop is to make FPGA and reconfigurable technology accessible to software programmers. Despite their frequently proven power and performance benefits, designing for FPGAs is mostly an engineering discipline carried out by highly trained specialists. With recent progress in high-level synthesis, a first important step towards bringing FPGA technology to potentially millions of software developers was taken.

The FSP workshop aims at bringing researchers and experts from both academia and industry together to discuss and exchange the latest research advances and future trends. This includes high-level compilation and languages, design automation tools that raise the abstraction level when designing for (heterogeneous) FPGAs and reconfigurable systems and standardized target platforms. This will in particular put focus on the requirements of software developers and application engineers. In addition, a distinctive feature of the workshop will be its cross section through all design levels, ranging from programming down to custom hardware. Thus, the workshop is targeting all those who are interested in understanding the big picture and the potential of domain-specific computing and software-driven FPGA development. In addition, the FSP workshop shall facilitate collaboration of the different domains.

Topics of the FSP Workshop include:

- High-level synthesis (HLS) and domain-specific languages (DSLs) for FPGAs and heterogeneous systems
- Mapping approaches and tools for heterogeneous FPGAs
- Support of hard IP blocks such as embedded processors and memory interfaces
- Development environments for software engineers (automated tool flows, design frameworks and tools, tool interaction)
- FPGA virtualization (design for portability, resource sharing, hardware abstraction)
- Design automation technologies for multi-FPGA and heterogeneous systems
- Methods for leveraging (partial) dynamic reconfiguration to increase performance, flexibility, reliability, or programmability
- Operating system services for FPGA resource management, reliability, security
- Target hardware design platforms (infrastructure, drivers, portable systems)
- Overlays (CGRAs, vector processors, ASIP- and GPU-like intermediate fabrics)
- Applications using HLS- or DSL-based approaches

We thank the authors who responded to our call for papers, the members of the program committee and the external referees who, with their opinion and expertise, ensured a very high quality program. Thank you all.

Tobias Becker
Frank Hannig
Dirk Koch
Daniel Ziener

Conference Organization

General Co-Chairs:

Tobias Becker, *Maxeler Technologies, UK*
Frank Hannig, *Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany*
Dirk Koch, *The University of Manchester, UK*
Daniel Ziener, *Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany*

Program Committee:

Hideharu Amano, *Keio University, Japan*
Jason H. Anderson, *University of Toronto, Canada*
Gordon Brebner, *Xilinx Inc., USA*
João M. P. Cardoso, *University of Porto, Portugal*
Sunita Chandrasekaran, *University of Houston, USA*
Andreas Koch, *Technical University of Darmstadt, Germany*
Miriam Leeser, *Northeastern University, USA*
Walid Najjar, *University of California Riverside, USA*
Gael Paul, *PLDA, France*
Marco Platzner, *University of Paderborn, Germany*
Dan Poznanovic, *Cray Inc., USA*
Rodric Rabbah, *IBM Research, USA*
Olivier Sentieys, *University of Rennes, France*
Dirk Stroobandt, *Ghent University, Belgium*
Gustavo Sutter, *Autonomous University of Madrid, Spain*
David Thomas, *Imperial College London, UK*
Kazutoshi Wakabayashi, *NEC Corp., Japan*
Markus Weinhardt, *Osnabrück University of Applied Sciences, Germany*
Peter Yiannacouras, *Altera Corp., Canada*

Reviewers:

Amano, Hideharu
Anderson, Jason H.
Becker, Tobias
Brebner, Gordon
Cardoso, João M. P.
Chandrasekaran, Sunita
Hannig, Frank
Höckmann, Rainer
Kenter, Tobias
Koch, Andreas
Koch, Dirk
Leeser, Miriam
Lösch, Achim
Najjar, Walid
Paul, Gael
Platzner, Marco
Poznanovic, Dan
Rabbah, Rodric
Sentieys, Olivier
Stroobandt, Dirk
Thomas, David
Wakabayashi, Kazutoshi
Weinhardt, Markus
Wiersema, Tobias
Yiannacouras, Peter
Ziener, Daniel

Application Acceleration with the VectorBlox MXP
[Keynote Talk]

Guy Lemieux
VectorBlox Computing Inc., Vancouver, Canada

Abstract

Many FPGA applications have internal data parallelism that can be sped up by a variety of techniques. In this talk, we will explore how we can speed them up with vector processing. In particular, using a counter vision application, we will demonstrate several features in the VectorBlox MXP processor that we have added to get massive speedup versus onboard ARM processors. This is done through a combination of algorithm adaptations, data size reductions, clever use of conditional execution, and the addition of custom instructions. We will demonstrate how a software compilation approach using soft processors in an FPGA can outperform hard processors.

Porting of a Particle Transport Code to an FPGA
[Keynote Talk]

Iakovos Panourgias
EPCC, University of Edinburgh, UK

Abstract

In this talk, we will discuss how software programmers can use recent advances for porting applications to FPGAs. We will discuss how writing code for FPGAs is hard. Using a particle transport code, we will show how a software programmer can create an FPGA port without writing a single line of VHDL and how to use performance models to estimate the runtime of the port. We will also show how minor algorithm modifications and data re-ordering and sizes affect performance of FPGA ports.

Allowing Software Developers to Debug HLS Hardware

Jeffrey Goeders and Steven J.E. Wilton
Department of Electrical and Computer Engineering
University of British Columbia, Vancouver, Canada
{jgoeders,stevew}@ece.ubc.ca

***Abstract*—High-Level Synthesis (HLS) is emerging as a mainstream design methodology, allowing software designers to enjoy the benefits of a hardware implementation. Significant work has led to effective compilers that produce high-quality hardware designs from software specifications. However, in order to fully benefit from the promise of HLS, a complete ecosystem that provides the ability to analyze, debug, and optimize designs is essential. This ecosystem has to be accessible to software designers. This is challenging, since software developers view their designs very differently than how they are physically implemented on-chip. Rather than individual sequential lines of code, the implementation consists of gates operating in parallel across multiple clock cycles. In this paper, we report on our efforts to create an ecosystem that allows software designers to debug HLS-generated circuits in a familiar manner. We have implemented our ideas in a debug framework that will be included in the next release of the popular LegUp high-level synthesis tool.**

I. Introduction

High-level synthesis (HLS) allows a program written in a software language (eg. C), to be automatically synthesized into a hardware circuit. HLS is quickly gaining popularity, particularly for FPGA programmable platforms, where it enables their use as compute accelerators alongside traditional processors [1], [2]. Both Altera and Xilinx have invested heavily in this technology, and we anticipate that HLS-based techniques may become the *dominant* design entry method for FPGAs in the future. Intel's recent announcement of their acquisition of Altera further emphasizes the shift of FPGAs from their glue-logic roots to a general-purpose algorithm acceleration platform. This shift will further increase the appetite for the fast turn-around design times and increased accessibility promised by HLS.

In order for HLS to deliver its promised benefits, a compiler is not enough. A complete ecosystem that provides the ability to *analyze, debug, and optimize* designs is essential. To be useful, this ecosystem has to be *accessible to software developers*. Software developers think of their systems in terms of sequential execution of instructions with limited explicit parallelism; this is in contrast to the actual FPGA implementation which consists of interconnected dataflow components operating in parallel across multiple clock cycles. As we will discuss in this paper, this disparity creates a chasm, that if not bridged, will significantly limit the effectiveness of analysis, debug, and optimization; this will, in turn, limit the suitability of HLS in designing most real systems.

Any debugging and optimization ecosystem is likely to be associated with some amount of on-chip instrumentation. In our previous publications, we have focused on the optimization of this instrumentation (especially the optimization of on-chip trace buffers) to maximize the amount of debugging information that could be provided to the user [3], [4] while minimizing overhead. In this paper, we take a step back and focus on the *user experience*. We discuss what sort of debug support we believe will make debugging HLS-generated circuits feasible for software developers, and describe our debug tool in which we have encapsulated some of these ideas.

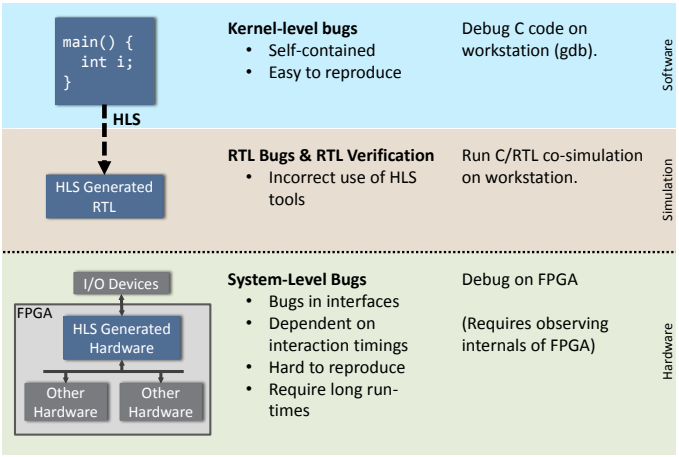


Fig. 1. Classification of bugs in an HLS system

Although we focus on functional debug in this paper, many of the ideas also apply to optimization (performance debug). The underlying challenges we will describe apply equally to both performance and functional debug, and our tool could be extended to support performance debug.

II. The Need for Hardware Debug

As shown in Figure 1, bugs can be classified into several categories, each of which can be addressed using a different debug flow. First, *kernel-level bugs* are errors in the algorithm specification (for example, errors in loop bounds, incorrect functions, or algorithmic errors). These bugs are typically confined to one module, and are often easy to reproduce (since, often, these bugs lead to incorrect behaviour every time the circuit is run). Often, these bugs can be identified by porting, compiling, and running the original C code directly on a workstation; mapping to hardware is not necessary. In tracking down these kinds of bugs, the designer can use software debug tools (eg. gdb, Eclipse) that are already familiar to software designers.

2nd International Workshop on *FPGAs for Software Programmers (FSP 2015)*, London, United Kingdom, September 1, 2015.

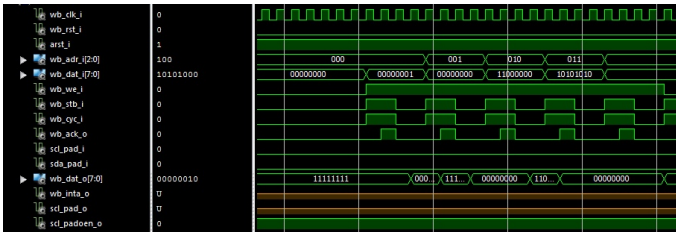


Fig. 2. Hardware view of a debug trace: Difficult for a software designer to understand!

A second class of bugs are those that appear in the generated RTL, even though the C code is correct. This may be caused by errors in the HLS tool itself, or errors in how the the HLS tool is used. FPGA vendors provide the ability to uncover these bugs using a co-simulation approach where the C and RTL code is simulated on a workstation [5]. Even if the code is correct, this level of system verification is essential; until HLS tools are fully mature, many designers will appreciate the confidence they achieve from a successful RTL simulation.

Despite extensive kernel-level and RTL simulation-level testing, there will always be some design errors that escape to the hardware implementation. There are at least three reasons this can happen. First, the software emulation will run much slower than the target hardware (typically 20 to 200 times slower [6]–[9]), limiting the thoroughness of tests that can be performed. In a complex system, it is impossible to completely test (or even enumerate) all corner cases. Second, this higher-level testing will not uncover problems related to interactions with the environment or with other modules in the system, yet this is where we expect many bugs to occur. In many large systems, HLS blocks are interfaced to legacy blocks designed using RTL techniques; these interfaces may be misunderstood by the software designer leading to subtle errors that are difficult to track down. Third, the environment in which the HLS block is used (for example, the input data stream) may not be exactly as assumed during RTL simulation; inaccuracies in the model of the environment may lead to bugs that only show up when the block is connected to the real environment. For these reasons, we expect that many errors will escape simulation to the hardware design, and the only way to find these errors is to debug the system *in-situ*, running on an FPGA.

III. Challenges for Software Engineers

The previous section made the case that certain types of design errors can only be uncovered by running a hardware implementation of the design. Debugging at the hardware level, however, is difficult for a software designer. The primary challenge during debugging an executing hardware design is that of visibility; finding the root cause of observed incorrect behaviour requires an understanding of the internal operation of the system. However, while a system is running, only I/O ports can be observed; internal signals, which are likely to provide a lot more useful information, can not be directly observed. To address this, there are commercial tools such as ChipScope from Xilinx [10], SignalTap II from Altera [11], and Certus from Mentor graphics [12]. These tools record selected signals in on-chip memories (called *trace buffers*) during the execution of the chip; at the end of the run, these trace buffers can be interrogated, and the user can use this information to understand the operation of the design, and eventually uncover the root cause of incorrect behaviour.

The challenge with this approach is that these tools provide visibility that has meaning only in the context of the generated RTL hardware. A software designer typically would not have an understanding of the underlying hardware; in fact, this is the primary reason that HLS methodologies are able to deliver high design productivity. A software designer views a design as a set of functions, each consisting of sequential control-flow code, while the underlying hardware consists of dataflow components operating in parallel across multiple clock cycles. Figure 2 shows a screenshot of the output of one of these tools; in this example, the behaviour of signals is illustrated using waveforms, a concept likely unfamiliar to many software designers. Even if understanding a waveform diagram is not a barrier, there may not be a one-to-one mapping between signals in the waveform and variables in the original C code. Further, since the HLS tool typically reorders instructions and extracts fine-grained parallelism, it is often difficult for a software designer to recognize the order of events and relate them to the order of instructions in the original C code. All of these factors make it very difficult for a software designer to use these hardware-oriented tools.

IV. Previous Work

To our knowledge, there have been two other debugging tools produced for HLS circuits that allow source-level, in-system debug. The first such tool was presented in 2003 [13], and was designed to work in conjunction with the Sea Cucumber HLS tool [14]. However, the debugger and HLS tool both utilized the now obsolete JHDL framework and are no longer supported or available for download. The debugger allowed the user to step through the source code while the circuit running on the FPGA was executed one cycle at a time. It supported inserting breakpoints and inspecting source-code variables. Our debugger builds on some of the ideas from this work, and includes several new approaches, as explained in the next section.

The second debugging tool produced was the Inspect debugger [15] in 2014, and was designed at the same time as our work presented in [3] and [4]. Since that time we have worked with the authors to combine the ideas from their work with ours into a single tool, which will be included in the next release of the LegUp high-level synthesis tool [16].

Other work presented in [17], [18] has focused on adding debug instrumentation to HLS circuits but does not include a debugger tool.

V. Our Debug Framework: HLS Scope

In this section, we describe how we are addressing the challenges faced by software designers debugging HLS circuits. We make our exposition concrete by presenting details of the debug tool we have created, and relate each of our ideas into features of this tool. Through this discussion, we will show that it is indeed possible to create a tool that resembles a software debugger, yet can be used to debug hardware designs running on an FPGA.

Figure 3 shows a screenshot of our tool. In the following discussions, we will refer to specific aspects of this diagram.

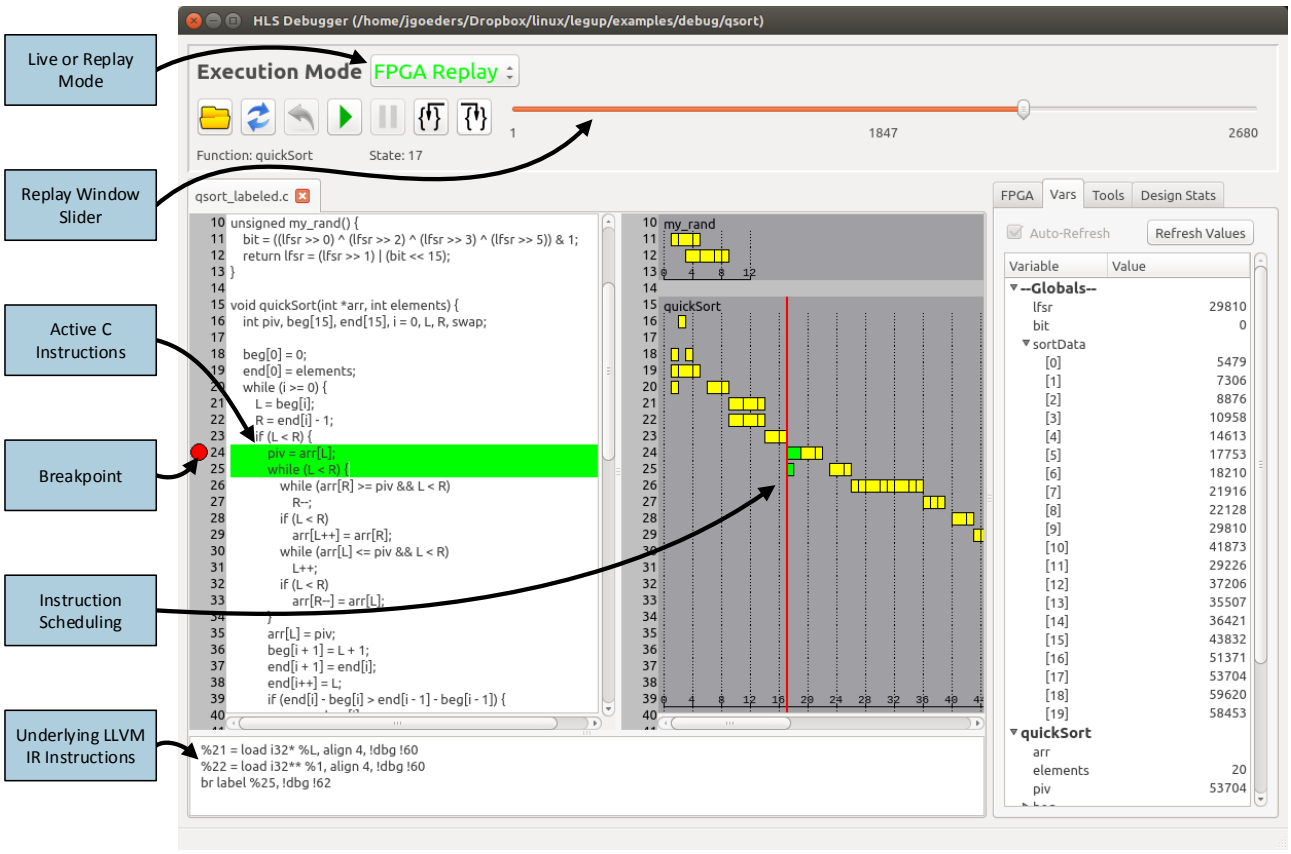


Fig. 3. Our framework: hardware debugger that a software engineer can understand

To make the debugging ecosystem easily accessible to software designers, we believe that any debug tool must resemble debug tools that these designers already understand (eg. gdb, Eclipse).[1] This means that users should be able to think of the design in terms of lines of code, and the state of the system as C-level variables. As shown in Figure 3, we achieve this in our tool; the source-level code is shown to the user (left panel), along with the C-level local and global variables (right panel).

In the left panel, the currently executing lines of source code are highlighted. The parallel nature of the circuit means that multiple C instructions can execute simultaneously. When this occurs we highlight multiple lines, as shown in Figure 3. This technique was also used in [13].

A. Gantt Chart

As shown in Figure 3, our tool presents a Gantt chart that shows the execution of each line of code over time. This provides much of the same information as a waveform diagram, in what we believe is a more software-friendly way. The diagram provides a mechanism for the designer to understand the fine-grained parallelism that has been uncovered by the HLS tool. As an example, in Figure 3, the assignments to $beg[0]$, $end[0]$, and the initialization operation of the while loop are started during the same cycle, and this is shown graphically in the Gantt chart. Similarly, some instructions may take more than one control step (in the hardware, this corresponds to more than one clock cycle, but a clock cycle is abstracted as a control step in our tool). Long operations such as divides, or instructions that operate on arrays, typically take more than one control step; examples of the latter can be seen in Figure 3. The boxes of the Gantt chart represent individual instructions of the underlying intermediate representation (IR), which is explained further in Section V-D.

Note that in the presence of compiler optimizations, the Gantt chart may not be as straightforward as that in Figure 3; we discuss the impact of compiler optimizations in Section V-E.

B. Debug Modes

Software designers expect to able to set breakpoints and single-step their design, and are accustomed to full visibility into the value of any variable at any point in the program. Yet, as described above, finding certain types of bugs requires running the circuit at-speed in-system. Providing enough infrastructure (trace buffers and associated logic) to provide full visibility into all signals in a hardware design running at-speed would require too much overhead to be practical. To address this, we have implemented two different debug modes: (1) *live mode*, in which the user can have full visibility, but does not run the circuit at speed, and (2) *replay mode*, in which the user can run the circuit at speed, but only has full visibility for a portion of the execution. Each of these is described below.

1) Live Mode: In the live mode, the system operates very much as a software debugger. The user can create breakpoints (limited by the number of hardware breakpoint units included

[1]Indeed, in future versions of the tool, we may investigate how we can integrate our techniques into Eclipse rather than providing a separate tool.

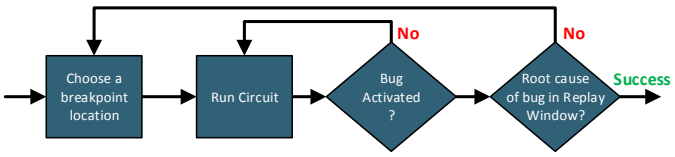


Fig. 4. Using Multiple Debug Iterations to expand the Replay Window

in the instrumentation) and single step through the code. Upon hitting a breakpoint, or completing a single step, the tool disables the controlling finite state machine (FSM), essentially pausing the system. While paused, the user can retrieve the value of all variables stored in on-chip memories (right pane of Figure 3).

2) Replay Mode: Debugging using the Live Mode involves starting and stopping the circuit during debug, similar to the technique used in [13]. This is not conducive to running the circuit at-speed. The Replay Mode provides the ability to run the circuit at-speed while preserving a software-like debug experience. We believe that the Replay Mode is likely the most important feature in our framework and best illustrates how the software and hardware worlds can be bridged.

This mode operates as follows. While in Live Mode, the user can set a breakpoint and run the circuit to the breakpoint. As the circuit runs, instrumentation added to the circuit records the changes to signals as well as the control flow executed by the program (these values are stored on-chip in trace buffer memories). After the program hits the breakpoint, the values in the trace buffer are read into the debug tool, and the user can enter the Replay Mode. While in Replay Mode, the user can still single-step and set breakpoints as before, however, all variable values and control flow information is obtained from the trace buffer data rather than the live values from the chip. In this way, the user can observe what the chip did during the at-speed run, while maintaining a software-like debug interface. In addition to single stepping, the user can use a slider to move ahead further in the buffer, as shown in the top-right of Figure 3. Interestingly, the slider can also be used to step backwards in time, providing the illusion of running the chip backwards. We anticipate that this feature will be important as users wish to "work backwards" to determine the root cause of unexpected behaviour. In fact, the technique of working backwards is already used in the software domain; examples include gdb's *Reverse Debugging* and Microsoft Visual Studio's *IntelliTrace*.

Note that the Live Mode requires on-chip trace buffers and associated logic to be added to the circuit. Because on-chip resources are limited, we can only store data for a limited number of instructions; we refer to the length of code for which data can be stored as the *Replay Window*. Within the replay window, we can provide a complete control-flow trace, allowing the user to observe which instructions are active for each execution step. Any variables that are updated within the replay window are available for inspection, after the point they are updated. Their value is unknown prior to the first update within the replay window. To handle the case where a variable is never updated during the replay window, we include instrumentation to provide the debugger with access to the memory controllers in the circuit. This allows us to read the value of a variable directly out of memory. While in Replay Mode, the user can step forwards and backwards through all instructions in the Replay Window, but can not step outside it. If the user wishes to go outside the Replay Window, multiple debug iterations are required as shown in Figure 4.

In our previous publications [3], [4], we present efficient buffer structures as well as methods of effectively compressing control flow and data information before storing it in the buffer. Those papers show that it is possible to record, on average, control and data information for 4322 lines of C code for each 100Kb of on-chip memory used for trace buffers. We could further reduce this by only recording selected signals or using off-line reconstruction methods; however, we have not yet investigated this further.

We refer the readers to [3], [4] for further details on the debug instrumentation, including data on execution trace length, area overhead, and impact on operating frequency.

C. Instruction-Level Parallelism

It is important to note that there is a significant difference between single step in our framework and single step in a software debugger. Single step in a software debugger will typically advance one source code statement. A single step in our framework may advance through multiple instructions at the same time, if those instructions are mapped to hardware that execute in the same clock cycle. As an example, in Figure 3, if the system is advanced one "instruction" beyond the red vertical line, both the assignment to piv and part of the $(L < R)$ check will be performed. Because this is hardware, and both are mapped to the same clock cycle, it is not possible to decouple these two operations and only single step through one of them. It may be possible to address these types of situations by creating a tool that modified the user circuit to either temporally separate these two operations or provide selective gating to each operation, however, that would result in significant changes to the user circuit, meaning the circuit being debugged may be very different than the original circuit.

In [13] the authors explored another approach. They provided virtual serialization, making it appear to the user that the statements were executed serially, when in reality they were executed in parallel in the hardware. Indeed this makes the debugger behave more like a traditional software debugger; however, this hides the parallelism from the user, and prevents them from restructuring their C code to explore different fine-grained parallelism optimizations.

D. IR Instructions

As shown in Figure 5, most HLS compilers compile software to hardware in several stages. First, C-to-C transforms such as loop restructuring are often performed, to make the C code more amenable to acceleration. The code is then often converted to an Intermediate Representation (IR) which resembles assembly language; several IR instructions are typically associated with each C operation. Optimizations are then

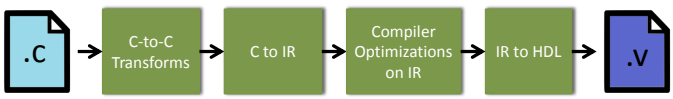


Fig. 5. Compilation flow in a typical HLS tool, illustrating the importance of the Intermediate Representation (IR)

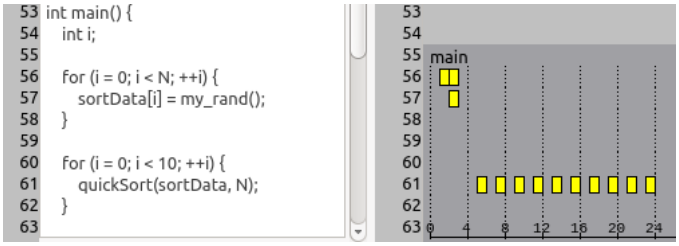


Fig. 6. Gantt chart showing loop unrolling optimization.

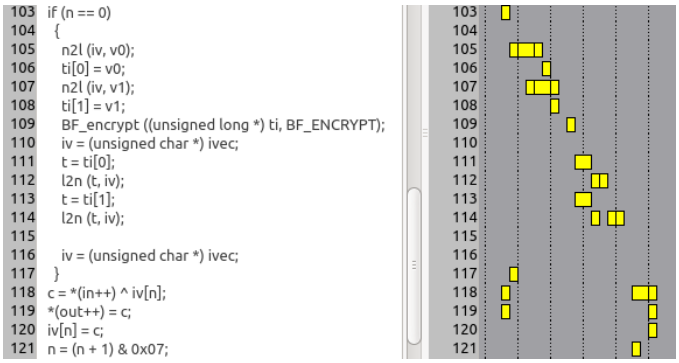


Fig. 7. Gantt chart showing code reordering optimization

performed and hardware is constructed directly from the IR representation.

Ideally, the software designer should be sheltered from the IR, and should not have to know of its existence. This is similar to how someone developing software should not have to know about the underlying assembly language of their compiled software. We anticipate that there are several reasons, however, that the HLS designer might want to inspect and understand the IR. First, C instructions often take multiple control steps, and it may sometimes become important to understand when primitive operations occur (this may be especially important when debugging multi-threaded applications). Second, during performance optimization, it may become important to understand why certain instructions take longer to execute than others; this information could be used to restructure the C code to lead to different fine-grained parallelism optimizations. Third, this information could be used by HLS tool vendors to help them as they optimize and debug their own compilers. For all these reasons, we have elected to expose the IR to the user. As shown in Figure 3, the bottom panel of the screen shows the IR for the executing instructions.

E. Compiler Optimizations

Most HLS tools provide the ability to select the level of optimization applied to the code before hardware is generated. For HLS tools built around the LLVM framework, the user can specify the familiar -O0, -O3, etc. flags. The higher the level of optimization, the more restructuring of the code that is performed before hardware is built.

Debugging optimized code (such as generated by the -O3 flag) is notoriously difficult. Because of this, when designing software, it is common practice to debug using the -O0 (unoptimized). We believe that this strategy does *not* work well for HLS-generated circuits for two reasons. First, changing the level of optimization will significantly change the timing of the resulting circuit. Since many of the bugs we anticipate are in the interfaces between blocks, changing the timing may result in significantly different behaviours. Thus, we believe that if the "-O3" version of a circuit is going to be "shipped" it is important to debug the "-O3" version of a circuit directly. Second, in many cases, we have found that compiling with "-O0" leads to much larger circuits, some of which would not fit on the target FPGA, making on-chip debugging impossible. This is different than software, in which as long as the program fits in (virtual) memory, it can be debugged.

Because of these reasons, we have designed our tool such that it can be used to debug optimized circuits. This has two implications. First, when optimizing circuits, variables are sometimes stored as registers in the datapath rather than in local memories. This requires a change in the instrumentation (as is discussed in [4]), however, it does not necessarily impact the user experience (unless variables are optimized away all together). Second, it means that the temporal relationship between various instructions may vary greatly; the result of a single-step may not be intuitive, since several instructions that are not together in the original program are executed out-of-order or simultaneously. We believe that it is important to provide the user this level of visibility (rather than abstracting the sequencing to program order) since understanding the *actual* order of execution of instructions may be very important when debugging block-level interfaces or other timing-related problems.

Figures 6 and 7 provide examples of how the Gantt chart aids users in debugging optimized code. In Figure 6 a sorting operation is called 10 times in a loop. The optimizing compiler has completely unrolled the loop and replaced the code with 10 subsequent calls to the function. In Figure 7, the compiler has performed code reordering optimizations. In this case, the two instructions immediately after the *if* block are independent of the contents of the *if* block and are executed before it. In both cases the Gantt chart helps the user in understanding the optimization.

VI. Extending our Framework

Although our debugging tool has been designed for use with the LegUp tool, we have designed it in a modular fashion, such that it can be extended to support other HLS tools, or expanded to explore new techniques for debugging HLS circuits. Figure 8 provides a diagram of the software organization of our tool.

At the heart of our tool is the *Debug Manager* which coordinates the debugging session. It tracks the current state of the design, and provides an API to control and observe the design. It generates signals when events occur in the circuit, such as when the state of the circuit changes or a breakpoint is encountered. The debug manager provides a *Backend API*, which allows for multiple backends to be added to the system to support different execution devices. For example, initially we supported a *Live* mode, which interacts with the FPGA, and a *Replay* mode, which uses the values from the trace buffers (both were previously described in this paper). Using the backend API we were able to very easily add a third execution mode, simulation-based execution using Modelsim. The backend API abstracts away the details of the device from the debugger tool. This abstraction could be used to test out

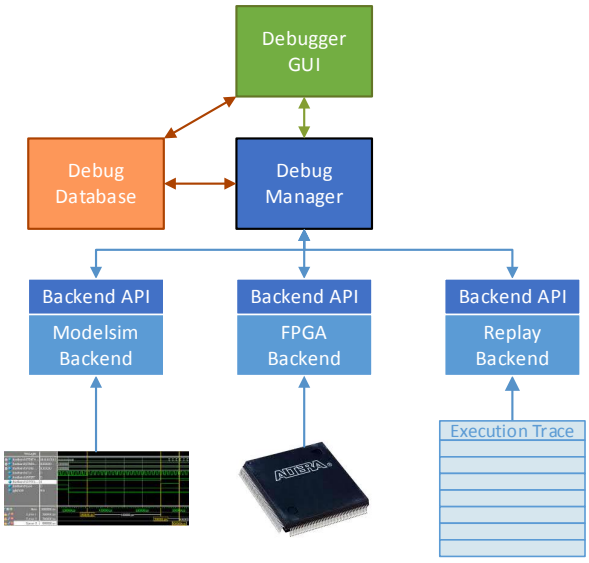


Fig. 8. Organization of our HLS-Scope debugger tool.

different techniques of debug instrumentation, or circuits from different HLS tools, without needing to modify other parts of the debugger tool.

The *Debugger GUI*, written in Python+Qt, provides a visual overlay on top of the Debug Manager. It issues requests to the Debug Manager, such as single-stepping, obtaining the current state of the circuit, or reading variable values. The GUI could be replaced with a different tool, such as a command-line interface, binary library, or Eclipse plug-in without needing to modify the rest of the system.

The final piece of the system, and perhaps the most essential is the *Debug Database*. This is a MySQL database that contains the details of the user's design, and was adapted from that in [15]. It is automatically populated during the HLS synthesis process. It keeps track of entities in the source code (functions, lines of C code, IR instructions, variables, data types, etc.), entities in the produced circuit (modules, FSM states, signals, memories, etc.), and the relationship between them. To port our debugger to another HLS tool, it would be necessary to modify the HLS flow to populate this database.

Currently, the debugger software connects to the FPGA via UART; however, in both the instrumented hardware and the debugger tool, the UART logic is modularized, allowing it to be replaced with other communication methods.

One limitation of the current tool is that it is limited to single-threaded software; although it handles fine-grained instruction-level parallelism, it does not support coarse-grained, thread-level parallelism that is emerging in the latest HLS tools. We plan to address this in future work.

VII. Conclusions

High-level synthesis is emerging as a mainstream design methodology, allowing software designers to target hardware implementation. As part of the HLS design process, software designers need the ability to debug their hardware systems, using debugging tools and methods familiar to them. In this paper we have presented HLS-Scope, our source-level debugger for the LegUp HLS tool. This tool is targeted to software designers, and provides a familiar debug interface, allowing them to single-step through their source code, place breakpoints and inspect variables. We include additional features such as a Gantt chart of the HLS scheduling and information of the underlying IR instructions to help bridge the gap between the sequential software and the optimized, parallelized circuit. The debug tool is open-source, modularized for extension to other applications, and available in the next release of LegUp.

References

[1] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, April 2011.

[2] W. Meeus, K. Van Beeck, T. Goedem, J. Meel, and D. Stroobandt, "An overview of todays high-level synthesis tools," *Design Automation for Embedded Systems*, vol. 16, no. 3, pp. 31–51, 2012.

[3] J. Goeders and S. J. Wilton, "Effective FPGA debug for high-level synthesis generated circuits," in *International Conference on Field Programmable Logic and Applications*, Sep. 2014, pp. 1–8.

[4] ——, "Using Dynamic Signal-Tracing to Debug Compiler-Optimized HLS Circuits on FPGAs," in *International Symposium on Field-Programmable Custom Computing Machines*, May 2015.

[5] K. Wakabayashi, "CyberWorkBench: integrated design environment based on C-based behavior synthesis and verification," in *International Symposium on VLSI Design, Automation and Test.*, April 2005, pp. 173–176.

[6] K. Tsoi, K. Lee, and P. Leong, "A massively parallel RC4 key search engine," in *Symposium on Field-Programmable Custom Computing Machines*, Apr. 2002, pp. 13–21.

[7] J. Beeckler and W. Gross, "FPGA particle graphics hardware," in *Symposium on Field-Programmable Custom Computing Machines*, Apr. 2005, pp. 85–94.

[8] K. Whitton, X. Hu, C. Yu, and D. Chen, "An FPGA Solution for Radiation Dose Calculation," in *Symposium on Field-Programmable Custom Computing Machines*, Apr. 2006, pp. 227–236.

[9] J. Lee, L. Shannon, M. Yedlin, and G. Margrave, "A multi-FPGA application-specific architecture for accelerating a floating point Fourier Integral Operator," in *International Conference on Application-Specific Systems*, Jul. 2008, pp. 197–202.

[10] Xilinx, "ChipScope Pro Software and Cores: User Guide," Apr. 2012.

[11] Altera, "Quartus II Handbook Version 13.1 Volume 3: Verification; 13. Design Debugging Using the SignalTap II Logic Analyzer," Nov. 2013.

[12] Mentor Graphics. Certus ASIC Prototyping Debug Solution. http://www.mentor.com/products/fv/certus. Accessed: 2014-12-03.

[13] K. Hemmert, J. Tripp, B. Hutchings, and P. Jackson, "Source level debugger for the Sea Cucumber synthesizing compiler," in *Symposium on Field-Programmable Custom Computing Machines*, April 2003, pp. 228–237.

[14] J. L. Tripp, P. A. Jackson, and B. Hutchings, "Sea Cucumber: A Synthesizing Compiler for FPGAs," in *International Conference on Field-Programmable Logic and Applications*, 2002, pp. 875–885.

[15] N. Calagar, S. Brown, and J. Anderson, "Source-Level Debugging for FPGA High-Level Synthesis," in *International Conference on Field Programmable Logic and Applications*, Sep. 2014, pp. 1–8.

[16] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 13, no. 2, pp. 24:1–24:27, 2013.

[17] J. S. Monson and B. Hutchings, "New approaches for in-system debug of behaviorally-synthesized FPGA circuits," in *International Conference on Field Programmable Logic and Applications*, Sep. 2014, pp. 1–6.

[18] J. S. Monson and B. L. Hutchings, "Using Source-Level Transformations to Improve High-Level Synthesis Debug and Validation on FPGAs," in *International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 5–8.

Model-based Hardware Design for FPGAs using Folding Transformations based on Subcircuits

Konrad Möller*, Martin Kumm*, Charles-Frederic Müller†, Peter Zipf*
*Digital Technology Group, University of Kassel, Germany
†Volkswagen AG, Braunschweig, Germany
*Email: {konrad.moeller, kumm, zipf}@uni-kassel.de
†Email: charles-frederic.mueller@volkswagen.de

***Abstract*—We present a tool flow and results for a model-based hardware design for FPGAs from Simulink descriptions which nicely integrates into existing environments. While current commercial tools do not exploit some high-level optimizations, we investigate the promising approach of using reusable subcircuits for folding transformations to control embedded multiplier usage and to optimize logic block usage. We show that resource improvements of up to 70% compared to the original model are possible, but it is also shown that subcircuit selection is a critical task. While our tool flow provides good results already, the investigation and optimization of subcircuit selection is clearly identified as an additional keypoint to extend high-level control on low-level FPGA mapping properties.**

I. Introduction

The use of domain-specific modeling tools like Matlab/Simulink is a common way to describe (and test) data flow dominated applications, commonly denoted as model-based design. It has proven successful in automatic code generation which is the de-facto standard, e. g., in the automotive domain for many years. Up to 80% of the processor code in todays embedded control units is generated from Matlab/Simulink [1]. The increasing demand for processing high sample frequencies recently lead to performance requirements that exceed the capabilities of embedded CPUs. FPGAs provide a solution for this problem as they yield the required computational power. However, typical sample frequencies are still much lower than the FPGAs' system clock frequency. This opens the opportunity to reduce FPGA resources by computing parts of the design using time-multiplexing while sharing the computation modules.

A well known method to automatically transform a parallel data flow graph (DFG) into a sequential circuit is *folding* [2]. During the folding transformation, common operators like, e. g., multipliers are implemented only once and shared by using multiplexers and additional registers. These resources and the required controller introduce an overhead which has to be lower than the resources saved due to sharing to gain any benefit [3]. As the number of multiplexers directly scales with the number of inputs of shared operands, an overhead reduction could be obtained when operations are combined to larger subcircuits instead of equipping each single operation with multiplexers and registers. In principle, the same solution could be found with the right operator selection, scheduling and binding, but for this the right parameters have to be known. The use of common subcircuits instead removes multiplexers and registers per construction. A subcircuit as defined in this work corresponds to a subgraph of the DFG. The more frequently a subcircuit occurs, the more resources can be saved due to sharing. However, the larger a common subcircuit is, the smaller is typically its frequency of occurrence. In addition, several independent common subcircuits may exist in the design which may even partly overlap, leading to a large design space. The task is thus related to a subgraph partitioning problem based on sets of isomorphic subgraphs. The target function on the other hand is based on implementation costs which are not directly related to individual subgraphs or to partition properties like size or number. Besides this, subcircuits can be used to reach a resource optimal point in the design space which meets the throughput requirements.

The idea behind our investigation is to apply well known high-level transformations to the Simulink description targeting resource reductions at the lowest register transfer/FPGA level. We present a tool flow which automatically utilizes the folding transformation to share arbitrary common subcircuits and show the benefits for this approach by a design space exploration of several benchmark circuits. The main contribution of this work is an extensive analysis of the results which were generated during this exploration. Besides this, we show that the results obtained with our tool flow are always better in terms of slices than the folding transformations of the Matlab HDL coder [4], which was taken as state-of-the-art reference in this work.

II. Background

The *identification* of common subcircuits, also known as subcircuit recognition [5], subgraph enumeration [6]–[8] or clone detection [1], is a well known problem which appears in many different disciplines and is akin to the subgraph isomorphism problem which is known to be NP-hard [6]. Powerful methods have evolved in the last three decades. A good introduction into the topic can be found in [5]. However, the *beneficial use* of common subcircuits in synthesis is still not well understood. Common subcircuits have been used in high-level synthesis (HLS) tools targeting behavioral input languages like C [7]–[10]. A tool flow that enumerates subgraphs and re-uses them in the xPilot HLS tool was presented by Cong and Jiang [7]. They report FPGA resource reductions

2nd International Workshop on *FPGAs for Software Programmers (FSP 2015)*, London, United Kingdom, September 1, 2015.

by about 20% on average. The sharing of single operations as well as common subcircuits (called composite operators or patterns) within the high-level synthesis (HLS) tool LegUp is analyzed in [10]. The sharing is limited to two operations with non-overlapping life times, i. e., one physical unit is used to compute two operations in the algorithm when no additional registers are required to store intermediate results. A greater benefit is reported when common subcircuits are used instead of single operations. They also analyzed the impact of the FPGA architecture and obtained area reductions from 7 to 12% by using subcircuit sharing.

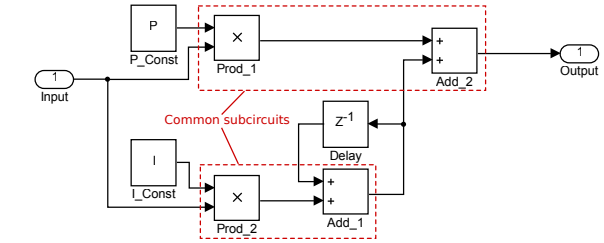
Common subcircuits also have been used in folding to reduce the computation time of the folding transformation [11]. There, the overall system as well as a common subcircuit are folded separately which is called hierarchical folding. The result is identical to the folding of the complete circuit for single operations, but reduces the complexity for M identical blocks from $\mathcal{O}(M^3)$ to $\mathcal{O}(M)$ (assuming the subcircuits are known in advance) [11]. A hierarchical synthesis methodology is also described in [9]. There, resource sharing is performed independently at each hierarchy level including controllers. To the best of our knowledge, common subcircuits were not used so far for resource reduction within the folding transformation.

III. Folding Transformation

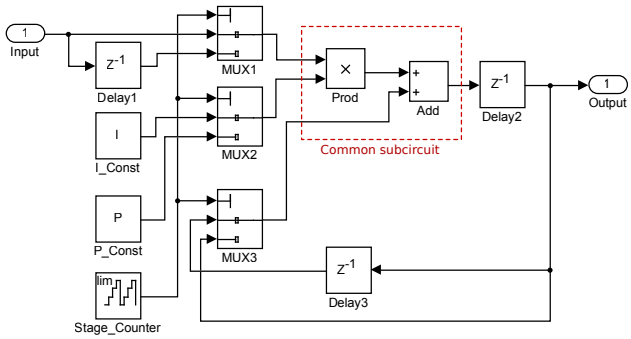
The folding transformation [2] is a systematic way to realize the time-multiplexed reuse of identical operations like e.g. additions and multiplications. In the implementation considered in this work, this can additionally be combinations of single operations to larger subcircuits which can be found more than once in the circuit. A common subcircuit or operation that is shared using folding is denoted *folding core* in the following. A circuit may consist of several common subcircuits which may be mutually exclusive, i. e., a subset of subcircuits may be selected as folding cores. Suitable common subcircuits used in the design space exploration are selected by the user in this work. This step could be automated by one of the subcircuit recognition methods discussed in [1], [5], [7], [8].

The folding procedure is illustrated by an example Simulink model of a discrete PI-controller as shown in Fig. 1(a). The possible folding cores could be a single product or a single addition, denoted as $\{Prod\}$ and $\{Add\}$ or the common subcircuit $\{Prod,Add\}$ as highlighted in Fig. 1(a). In the next step, a scheduling is required to determine the time step in which each subcircuit will be executed. This is important to provide the right input data at the right time in the time-multiplexed circuit. In the given example this could be $\{Prod\_2,Add\_1\}$ in the first time step and $\{Prod\_1,Add\_2\}$ in the second time step for the folding with common subcircuits. The minimal number of required time steps which leads to a valid solution is called the *folding factor* N. In the best case it is equal to the number of identical subcircuits.

The scheduled processing times can be verified by the help of the *folding equation*, which determines the delay D in number of clock cycles between two nodes U and V in the



(a) Original model



(b) Folded model

Fig. 1: Example of a PI controller described in Simulink

folded circuit:

$$D(U \xrightarrow{e} V) = Nw_e - P_u + v - u \geq 0 \tag{1}$$

where w_e is the delay in number of clock cycles of the edge e from nodes U to V in the original DFG, P_u is the latency of U and u and v are the scheduled execution times of U und V, respectively.

Now, all the common subcircuits can be replaced by their corresponding folding cores which are equipped with multiplexers at their inputs. Each multiplexer input may require a number of additional registers (D) as given by the *folding equation*. Note that this may lead to a more complex wiring, but in our experiments this was not a limiting factor. The resulting folded PI controller circuit using the folding core $\{Prod,Add\}$ is shown in Fig. 1(b).

To obtain a benefit from folding, the resulting circuit size S_f of the N times folded circuit has to be smaller than the size S_o of the original circuit. Both circuits can be separated into the resources of the folding cores ($S_{\text{folding core}}$), non-folded parts (S_{remain}) and the overhead due to folding (S_{overhead}), leading to the condition:

$$S_f < S_o \tag{2}$$

$$S_{\text{overhead}} + S_{\text{folding core}} + S_{\text{remain}} < NS_{\text{folding core}} + S_{\text{remain}} \tag{3}$$

$$S_{\text{overhead}} < (N-1)S_{\text{folding core}} \tag{4}$$

Clearly, the overhead has to be smaller than the size of the saved folding cores. Thus, selecting a large folding factor and a large folding core should be worthwhile. But the capability to maximize both is limited by the structure of the original circuit, so a tradeoff has to be found.

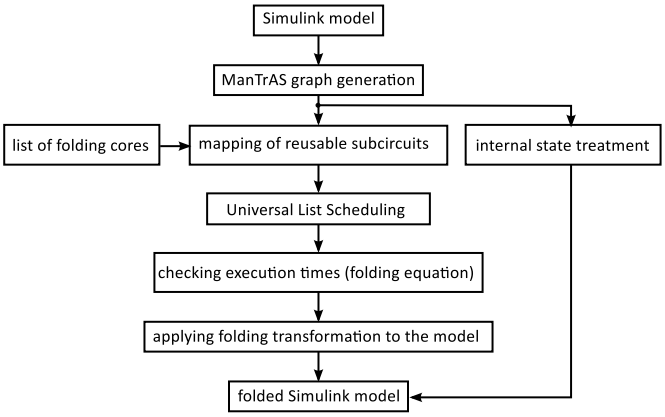


Fig. 2: Overview: Implemented folding transformation flow

IV. Transformation Flow

In this section, the transformation flow of the implemented automatic folding transformation tool is presented. We perform structural transformations at the Simulink level to get the folded result. An overview is given in Fig. 2. The input of the transformation flow is the Simulink model of the design which is to be optimized. In the first step the model is transformed into an object oriented Matlab data structure by a tool called ManTrAS (**M**atlab **an**alysis and **Tr**ansformation **A**PI for **S**imulink) [12]. It provides an easy way to work with the data representing a Simulink model in order to analyze the model and to do the required transformations. Moreover a ManTrAS graph can be transformed back into a Simulink model. After the back-transformation everything which could be done with the original Simulink model can be applied to the folded one. This includes the simulation using the original test bench, HDL code generation using Mathworks' HDL Coder and validation of the folded model which is an essential advantage of this approach.

The next step is the mapping of the folding cores to the corresponding blocks in the ManTrAS graph. This is done by putting all blocks which belong to one folding core into a Simulink subsystem which can be identified by having a specific type of folding core. In contrast to other approaches we do not only consider one type of folding core but also a set of different non-overlapping folding cores for the folding transformation.

Since the former parallel design will be processed in a sequential time-multiplexed way after the transformation, the folding cores have to be assigned to time steps. This is currently done by a general list scheduling with support of resource constraints and multi-cycle operations [13]. The resource constraints are used to limit the available folding cores to one per clock cycle to force the schedule to be valid for the time-multiplexing. The latency P_u of an operation is treated as a multi-cycle (blocking) operation in the scheduling. This guarantees that $v - u$ is always greater or equal than P_u and, thus, the folding equation result is always positive or zero. The corresponding delays are inserted at the corresponding multiplexer input. A single control unit (a mod-N counter) is inserted which controls the multiplexers to select the right input at the scheduled time step.

In addition to the original folding transformation, we support folding cores with internal states. For that, pipeline interleaving is applied to each folding core by simply replacing each register in the folding core by N registers [2], [11].

V. Experimental Setup

This section comprises an experimental design space exploration and evaluation of a benchmark set with different folding cores selected for the folding transformation. The results were automatically generated using the presented transformation flow. The evaluation is done with four commonly used applications in the domains control engineering and digital signal processing. The following applications have been chosen, implemented as functional Simulink models with a 32 bit fixed point precision and can be accessed online [14]:

A) 16 tap finite-impulse response (FIR) filter
B) Park-Clarke transformation (PCT)
C) Triple PID controller (TPID)
D) Infinite-impulse response (IIR) filter from [2]

While the FIR and IIR filters are well known, the Park-Clarke transformation is a combination of the alpha-beta transformation [15] and the direct-quadrature-zero transformation [16]. It is an important transformation in automotive controls and is used for the observing part of multiple-phase brushless DC motors. It consists of several sine look-up tables, additions/subtractions and multiplications.

The Triple PID controller is a design which implements three standard discrete PID controllers in parallel using the rectangular method for intergrals and differentials.

We distinguish between four implementation cases to compare the different folding strategies: The first case is the *original* unfolded design as reference. The second case is the *single operation folding*, which represents the folding strategy to share resource intensive single operations like, e. g., multiplications. The third case is the folding using *common subcircuits* as folding core, which was the main target of our exploration. The intention is mainly to show the benefits compared to the *single operation folding*. The HDL Coder resource sharing is taken as the last case, because it is a state-of-the-art commercial solution. The selected cores and their number can be found in TABLE I following the notation introduced in Sec. III. For example, in the FIR benchmark with folding factor $N = 5$ we reduce 5 cores each consisting of 2 delays, 2 products and 2 adders to 1 core and 2 cores consisting of 1 delay, 1 product and 1 adder to 1 core. This is denoted as 5{2delay,2prod,2add}, 2{delay,prod,add}. In some cases different folding cores are selected while the folding factor is the same, which results in different realizations for the same folding factor.

The folding transformation was performed for each selection using the presented flow, resulting in a folded Simulink model. This model was verified within the Simulink environment by a direct comparison to the input/output-behavior of the original unfolded model. After this step VHDL-Code was

TABLE I: Folding cores selected for the evaluation

appl.	folding cores			
	N	cores	N	cores
FIR	2	2{7delay,7prod,7add}	14	14{prod,add,delay}
	2	2{delay,prod}	14	14{prod,delay}
	3	3{4prod,4add, 4delay}, 2{2prod,2add,2delay}	15	15{prod}
	4	4{3prod,3add,3delay}, 3{prod,add}	15	15{add}, 15{prod}
	5	5{2delay,2prod,2add}, 2{delay,prod,add}	15	15{prod,add}
	7	7{2prod,2add,2delay}		
PCT	2	2{1conv,5sub, 3sin,3prod}	6	6{sin}, 6{prod}, 2{prod}, 6{sub}, 4{sub}
	2	2{sub,sin,prod}, 2{sub,sin,prod}, 2{sub,sin,prod}	6	6{sin}, 6{prod}, 2{prod}, 3{sub}, 3{sub}, 2{sub}, 2{sub}
	3	3{const,sub,sin,prod}, 3{sub,sin,prod}	6	6{sub,sin,prod}, 2{prod}, 4{sub}
	6	6{sin,prod}	6	6{sin}
	6	6{sin}, 6{prod}, 2{prod}	6	6{sub,sin,prod}
TPID	3	3{sub,prod,add}	9	9{prod}
	3	3{single PID}	9	9{prod}, 9{add}, 6{sub}
	3	3{P}, 3{I}, 3{D}	9	9{prod,add}
	6	6{sub,prod}		
IIR	2	2{prod,add,add}	4	4{add}, 4{prod}
	2	2{prod,add,prod,add}	4	4{prod}
	2	2{prod,add}		

generated using HDL Coder (v2.2) and functionally verified with ISim. Finally the VHDL code was synthesized for a Virtex 4 FPGA (xc4vlx200-10-ff1513) to get the required slice and DSP block count as well as timing information. The settings for DSP usage where set to *Auto*. The HDL Coder was used with different resource sharing factors which can be provided by the user to get a resource-latency tradeoff.

Special care has to be taken when constant values are included in the design. In such cases the constants may have an impact on the resulting design size, because different optimizations can be performed by the tools during the synthesis process. As we want to consider the general case we prevented the constants from being trimmed by replacing them by external inputs before synthesis. This means that the results in TABLE II and Fig. 3 and 4 represent an upper bound for the design sizes as for specific constants (like power-of-two values), less resources are possible.

VI. Results

A summary of the results can be found in TABLE II. The required resources of the unfolded (*original*) design are compared to the resources of the solution generated with the proposed folding flow using common subcircuits or single operation folding, which lead to the largest slice and DSP block reduction (*best fold.*) for the specific benchmark. The corresponding data points of these and the worse solutions can

TABLE II: Comparsion between original and best folded design and between common subcircuit and corresponding single operation folding for the example designs

	FIR filter		PCT		Triple PID		IIR filter	
	Slices	DSPs	Slices	DSPs	Slices	DSPs	Slices	DSPs
original	663	64	12910	66	3492	96	135	16
best fold.	485	6	3938	19	1319	40	393	8
savings (%)	*26*	*91*	*70*	*71*	*62*	*58*	*-191*	*75*
single op.	1474	4	3938	19	1984	16	621	4
comm. sub.	485	6	4242	19	1731	16	428	4
savings (%)	*67*	*-50*	*-7,7*	*0*	*13*	*0*	*31*	*0*

be found in Fig. 3. Moreover, the best result using common subcircuits (*comm. sub.*) is compared to the result of the corresponding single operation folding (*single op.*), as the investigation of their relation is the motivation of our work. A general observation is that a large amount of slice resources and DSP blocks can be saved compared to the original model and that the savings of common subcircuit folding typically surpass the savings of single operation folding.

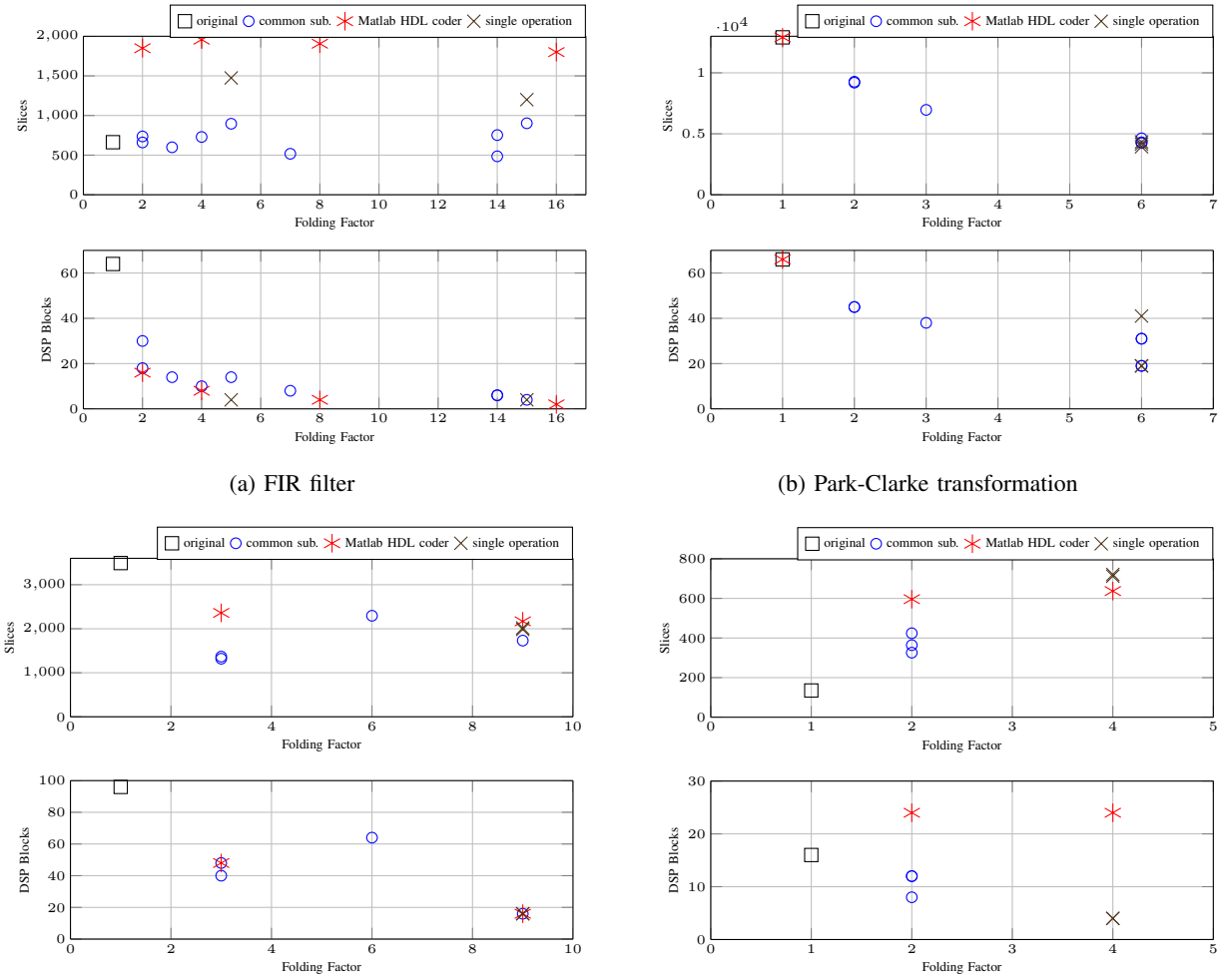
In the following subsections specific observations for the different benchmarks and figures of the explored design space are provided (all numbers refer to TABLE II).

A. 16 Tap FIR Filter

The results for the different FIR solutions can be found in Fig. 3 (a). For this very regular design the best solution can be achieved by a folding factor of 14 for the number of required slices and by a folding factor of 16 for the number of required DSP blocks. In order to achieve a slice reduction, the use of the largest common subcircuit (14{prod,add,delay}) is beneficial. The fact that the largest folding factor is leading to the smallest number of required DSP block is a general observation which holds for all analyzed benchmark circuits (A-D). For the FIR benchmark, there are many cases, especially with single operation folding, in which folding is leading to a much higher resource consumption compared to the unfolded design. This is the result of a large overhead compared to the saving which can be achieved by resource sharing for this rather small design. The best folded solution saves 26% of the required slices and 91% of the required DSP blocks compared to the original design. In the case of the FIR filter, common subcircuit folding is the only way to save slice resources.The corresponding single operation folding is not beneficial as the slice overhead is about 1000 slices, which exceeds the slice count of the original model.

B. Park-Clarke Transformation

In the Park-Clarke Transformation example, the best solution can always be found with the largest folding factor. The analyzed cases have an almost identical slice consumption for identical folding factors as it can be seen in Fig. 3 (b). The HDL Coder was not able to perform any resource sharing independent of the given sharing factor. By application of our transformation flow on this rather large



(a) FIR filter

(b) Park-Clarke transformation

(c) Triple PID controller

(d) IIR filter

Fig. 3: Results for slice and DSP block usage for all benchmarks

example we could achieve savings of about 70% for slice and DSP usage. The best solution for the single operation (6{sin},6{prod},2{prod},6{sub},4{sub}) and common subcircuit (6{sub,sin,prod},2{prod},4{sub}) folding have nearly the same slice consumption. This results from nearly identical folded solutions. In one case we define the common subcircuit (6{sub,sin,prod}) and in the other case each component of this common subcircuit is selected as a single operation (6{sin},6{prod},6{sub}). Based on a good scheduling for the single operation case, the architecture of the common subcircuit is reconstructed automatically during the transformation flow. The multiplexers in the single operation case thus have the same input at each port and can be replaced by a wire during synthesis.

C. Triple PID Controller

The results for the Triple PID controller can be seen in Fig. 3 (c). The best case in terms of slice usage is not the case with the largest or nearly largest folding factor for this example. This can be explained by the fact that very large folding cores can be found in the best cases with only one input, leading to only one input multiplexer. The low overhead on the one side is further enhanced by the large resource saving on the other side, because of the large folding cores. In this case the folding core size was the defining element in the tradeoff between folding factor and folding core size (cf. (4)). The comparison of single operation (9{prod}) and the corresponding common subcircuit (9{prod,add}) folding in TABLE II shows again that it is beneficial to search for larger common subcircuits rather than folding around single operations.

D. IIR Filter

The last example consists of only four multiplications and four additions and supplies only few folding alternatives. The results are plotted in Fig. 3 (d). The unfolded design is always the best solution in terms of slices, but choosing the multiplication and addition as folding core can significantly reduce the required DSP blocks. The HDL Coder resource sharing result is only able to save one out of four multipliers which leads to a larger slice overhead for time-multiplexing and no savings in DSP block consumption.

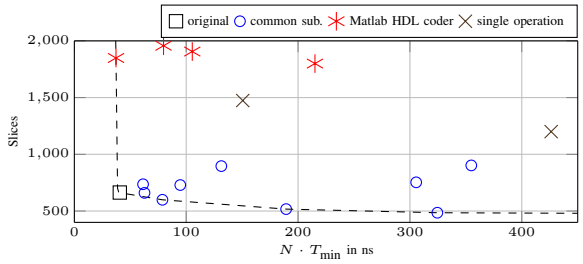


Fig. 4: Slice resource usage over $N \cdot T_{\text{min}}$ of calculation for the FIR filter example and pareto front (dashed line)

E. Resource Performance Tradeoff

With a set of different folding cores, leading to different folding factors and different latencies for data sample processing, our tool can deliver a tradeoff between resource consumption and latency of the processing of one data sample. The evaluated data points of the FIR filter example were used in Fig. 4 to show the number of required slices over the computation time $N \cdot T_{\text{min}}$, where T_{min} denotes the minimal clock period obtained from the timing analysis. A designer with specific latency or area limitations could pick the best possible solution very easily. For a sample period requirement the best solution in terms of slice usage can be found as the point at the very bottom which is left of the latency limit and for a slice limitation the best solution in terms of latency is the leftmost point which is lower than the slice limit.

F. Summary

The experimental results show that it is beneficial in terms of slice reduction to use common subcircuits instead of single operations as folding cores. Besides the slice reduction the selection of common subcircuits always leads to the best results if low latency is required. The largest DSP block reduction can be achieved by selecting the maximum folding factor in all cases, which can lead to a large slice overhead on the other hand. The relation of overhead and reduction by folding factor and/or folding core size (4) could be seen in some cases, but further investigations have to be done. The subcircuit selection itself is a critical task and it has a significant impact on the resulting design. However, the design space exploration which is possible with the presented transformation flow can help to find the best solution in order to fulfill the application constraints. The design space exploration considered the folding factor as well as the folding core size. A change in the degree of sharing, i.e., varying the amount of available folding cores was not analyzed, but should be considered in future work, as it delivers an additional optimization possibility.

VII. Conclusion

We presented an automated high-level transformation for Simulink models targeting optimized FPGA implementations. The resulting models can be used by standard tools to generate VHDL code. We showed results only for four models but it becomes evident that very large improvements can appear, up to 70% in our examples. Using the tool flow, a design space exploration concerning the folding factor and the folding core inputs could be established and first results were analyzed in this paper. The results clearly show the benefits of our approach but also, that the selection of folding cores is a key factor for the improvement of FPGA resource requirements for any specific model: while the usage of embedded multipliers can be directly controlled, logic block requirements depend largely on the selected subcircuits.

Currently, different core combinations are selected by the user, but the results indicate that there is a non-trivial dependency between the model structure, the folding factor and the subcircuits and their possible combinations. An algorithm for subcircuit selection and combination is obviously necessary to fully automize the process of defining reasonable folding cores as input to our tool flow. The development of such an heuristic and the definition of according selection criteria are therefore the main targets of our future work.

References

[1] F. Deissenboeck, B. Hummel, E. Jurgens, B. Schatz, S. Wagner, J. F. Girard, and S. Teuchert, "Clone detection in automotive model-based development," in *Software Engineering, 2008. ICSE '08. ACM/IEEE 30th International Conference on*, 2008, pp. 603–612.

[2] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*. John Wiley & Sons, 1999.

[3] W. Sun, M. J. Wirthlin, and S. Neuendorffer, "FPGA Pipeline Synthesis Design Exploration Using Module Selection and Resource Sharing," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 2, pp. 254–265, 2007.

[4] G. Venkataramani, K. Kintali, S. Prakash, and S. van Beek, "Model-based hardware design," in *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*. IEEE, 2013, pp. 69–73.

[5] N. Rubanov, "A High-Performance Subcircuit Recognition Method Based on the Nonlinear Graph Optimization," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 25, no. 11, pp. 2353–2363, 2006.

[6] J. L. White, M. J. Chung, A. S. Wojcik, and T. E. Doom, "Efficient algorithms for subcircuit enumeration and classification for the module identification problem," in *International Conference on Computer Design (ICCD)*. IEEE, 2001, pp. 519–522.

[7] J. Cong and W. Jiang, "Pattern-based behavior synthesis for FPGA resource reduction," in *Proceedings of the 16th international ACM/SIGDA symposium on FPGAs*, New York, USA, Feb. 2008, pp. 107–116.

[8] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 473–491, 2011.

[9] O. Bringmann and W. Rosenstiel, "Resource sharing in hierarchical synthesis," in *Computer-Aided Design, IEEE/ACM International Conference on*. IEEE, 1997, pp. 318–325.

[10] S. Hadjis, A. Canis, J. H. Anderson, J. Choi, K. Nam, S. Brown, and T. Czajkowski, "Impact of FPGA architecture on resource sharing in high-level synthesis," pp. 111–114, 2012.

[11] K. K. Parhi, "Hierarchical Folding and Synthesis of Iterative Data Flow Graphs," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 60, no. 9, pp. 597–601, 2013.

[12] C. Kolassa, D. Dieckow, M. Hirsch, U. Creutzburg, C. Siemers, and B. Rumpe, "Objektorientierte Graphendarstellung von Simulink-Modellen zur einfachen Analyse und Transformation," *Tagungsband AALE 2013, 10. Fachkonferenz*, pp. 277–286, 2013.

[13] G. De Micheli, *Synthesis and Optimization Of Digital Circuits*. McGraw-Hill, 2003.

[14] K. Möller, M. Kumm, and C.-F. Müller, "Simulink Benchmarks and Results," http://www.uni-kassel.de/go/simulink_benchmarks, 2015.

[15] E. Clarke, *Circuit Analysis of AC Power Systems*. Wiley & Sons, 1943.

[16] R. H. Park, "Two Reaction Theory of Synchronous Machines Generalized Method of Analysis-Part I," *AIEE Transactions*, vol. 48, pp. 716–727, 1929.

Automatic Nested Loop Acceleration on FPGAs Using Soft CGRA Overlay

Cheng Liu, Ho-Cheung Ng and Hayden Kwok-Hay So
Department of Electrical and Electronic Engineering
The University of Hong Kong
Email: {liucheng, hcng, hso}@eee.hku.hk

***Abstract*—Offloading compute intensive nested loops to execute on FPGA accelerators have been demonstrated by numerous researchers as an effective performance enhancement technique across numerous application domains. To construct such accelerators with high design productivity, researchers have increasingly turned to the use of overlay architectures as an intermediate generation target built on top of off-the-shelf FPGAs. However, achieving the desired performance-overhead trade-off remains a major productivity challenge as complex application-specific customizations over a large design space covering multiple architectural parameters are needed.**

In this work, an automatic nested loop acceleration framework utilizing a regular soft coarse-grained reconfigurable array (SCGRA) overlay is presented. Given high-level resource constraints, the framework automatically customizes the overlay architectural design parameters, high-level compilation options as well as communication between the accelerator and the host processor for optimized performance specifically to the given application. In our experiments, at a cost of 10 to 20 minutes additional tools run time, the proposed customization process resulted in up to 5 times additional speedup over a baseline accelerator generated by the same framework without customization. Overall, when compared to the equivalent software running on the host ARM processor alone on the Zedboard, the resulting accelerators achieved up to 10 times speedup.

I. Introduction

Offloading compute intensive nested loops to FPGA accelerators has been demonstrated by many researchers to be an effective solution for performance enhancement across many application domains [1], [2]. However, the relatively low productivity in developing FPGA-based compute applications remains one of the major obstacles that hinder widespread employment of FPGAs [3]. To address this challenge, a number of researchers have turned to the use of virtual FPGA overlay architectures built on top of the physical FPGA configurable fabric to help with improving design productivity through fast compilation, good design portability and debugging support [4], [5], [6], [7], [8], [9], [10], [11].

Despite the great advantages on design productivity, the additional layer on top of the physical FPGA inevitably introduces performance and resource consumption penalty. An overlay must ensure that the overall FPGA acceleration performance remains competitive. Otherwise, mapping the loop kernels to the overlay based FPGA accelerators will not be as useful. Therefore, the capability to customize the overlay specifically to an application or a domain of application becomes essential to the overlay based FPGA accelerator design. However, navigating through a labyrinth of architectural and compilation parameters to fine-tune an accelerator's performance is a slow and non-trivial process. To require a user to manually explore such vast design space is going to counteract the productivity benefit of the utilizing overlay in the first place.

We have been developing in-house a soft coarse-grained reconfigurable array (SCGRA) overlay based nested loop acceleration framework targeting a hybrid CPU-FPGA system called QuickDough, which allows rapid compilation from C loops to FPGA with a library of pre-built overlay bitstreams [10]. In this work, we mainly focus on automatically customizing the overlay architectural parameters, exploiting loop unrolling and hardware-software communication in combination with buffer sizing specifically to an application with given high-level resource constraints. In particular, by taking advantage of the regularity of the SCGRA overlay, a multitude of design metrics such as performance and hardware consumption can be accurately estimated using analytical models once the overlay scheduling result is available. While the overlay scheduling depends on much less design parameters, the overall customization framework can be dramatically simplified. With both the efficient application-specific customization and rapid compilation, the proposed design framework ensures both high design productivity and high performance of FPGA loop acceleration.

From our experiments, it took the framework 10 to 20 minutes to complete the loop accelerator customization using our proposed two-step approach, which was up to 100 times faster than an exhaustive search through the design space. With customization, the resulting accelerators performed up to 5 times faster than a corresponding baseline accelerator before customization. Overall, when compared to the performance of the benchmark executed on the host ARM processor, the resulting FPGA accelerators achieved up to 10× speedup.

II. Related Work

Overlay architecture which is a virtual intermediate architecture overlaid on top of off-the-shelf FPGA is increasingly applied as a way to address the productivity challenge.

Various overlays with diverse configuration granularities and flexibility ranging from virtual FPGAs [4], [6], [5], array-of-FUs [7], [8], [11], soft CGRA [9], [10], soft GPU [12], vector processors[13], [14] to configurable processors or multi-

2nd International Workshop on *FPGAs for Software Programmers (FSP 2015)*, London, United Kingdom, September 1, 2015.

core processors [15], [16], [17], [18], [19], [20] have been developed over the years. SCGRA overlay provides unique advantages on compromising hardware implementation and performance for compute intensive nested loops as demonstrated by numerous ASIC CGRAs [21], [22]. Most importantly, it allows both rapid compilation by taking advantage of the overlays' tiling structure [23] and efficient bitstream reuse within the design iterations of an application [10], thus it is particularly promising for high productivity nested loop acceleration.

In addition, customizing the CGRA specifically for an application or a domain of application provides promising performance improvement while saving the hardware resource at the same time as demonstrated in CGRA work targeting ASIC design [24], [25], [26]. While CGRA customization on ASIC is relatively limited due to the tape-out cost, CGRA overlays allow more intensive architectural customization providing just enough hardware to the target application or application domains because of the FPGA's inherent programmability. In [27], Coole and Stitt proposed to provide the overlay with limited flexibility instead of full configurability specifically to a group of design. With this customization, the area overhead was reduced significantly. The authors in [28] developed an SCGRA topology customization method using genetic algorithm and showed the potential benefits of the SCGRA overlay customization. Nevertheless, the rest of the system design parameters were not covered. In [2], the authors formalized the loop acceleration on a regular processing array overlay on FPGA. They focused on the hardware resource constrain, IO bandwidth constrain and the loop parallelism partition while processing architectural design parameters were not included. In order to achieve both high design productivity and high performance with low overhead, a complete nested loop acceleration framework targeting CPU-FPGA system is developed in this work. It supports intensive application-specific customization including the overlay architectural customization, the compilation customization and communication interface customization for optimized performance.

III. Nested Loop Accelerator Design Framework

By using a regular SCGRA overlay built on top of the physical FPGA devices, we have developed an automatic nested loop acceleration framework called QuickDough. QuickDough targets hybrid CPU-FPGA computing systems where the FPGA is devoted to accelerating compute intensive loop kernel and CPU handles the rest of the application. Figure 1 depicts an overview of the design framework, highlighting the complementary *accelerator generation* and *accelerator customization* paths.

By design, the steps along the accelerator generation path are short and essential during rapid design iterations. Collectively, they are able to generate FPGA loop accelerators making use of a pre-built bitstream library in the order of seconds [10].

Meanwhile, the focus of this paper is on the accelerator customization path, which is relatively slow but is necessary

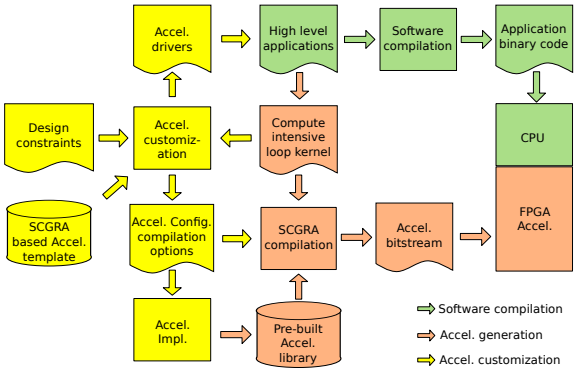


Fig. 1. Automatic nested loop acceleration framework

for improving performance of the resulting accelerators on a per-application basis. These steps automatically tunes the design parameters including overlay architectural parameters, compilation options as well as communication between the FPGA accelerator and host processor specifically to a user application under user constraints such as hardware resource budgets. With the customized design parameters, HDL models of the corresponding SCGRA overlay and their associated drivers are then generated. Afterwards, the drivers will be used by the software compiler while the FPGA accelerator will be implemented and stored in the accelerator library, which can be reused by the fast accelerator generation path in subsequent compilations.

A. SCGRA based FPGA accelerator

Figure 2 shows the design of a typical SCGRA overlay based FPGA accelerator. In the accelerator, on-chip memory i.e. IBuf and OBuf are used to buffer the communication data between the host CPU and the accelerator. A controller is also presented in hardware to control the operations of the accelerator as well as memory transfers. The SCGRA, which is the kernel computation fabric, consists of an array of PEs and it achieves the computation task through the distributed control words stored in each PE. The AddrBuf stores all the valid IO buffer accessing addresses of the computation. The current implementation of a PE template is also presented in this figure. At the heart of the PE is an ALU, which is supported by a multi-port data memory and an instruction memory. Data memory stores intermediate data during the computation while instruction memory stores all control words that determines the action of the PE. In addition, a global signal from the AccCtrl block controls the start/stop of all PEs in the array.

B. Loop execution on the FPGA accelerator

Figure 3 illustrates how the loop is executed on the FPGA accelerator. First of all, data flow graph (DFG) is extracted from the loop and then it is scheduled on to the SCGRA overlay based FPGA accelerator. Depending on how much the loop is unrolled and transformed to DFG, the DFG may be executed repeatedly until the end of the original loop. In addition, data transfers for multiple executions of the same DFG are batched into groups as shown in Figure 3. On the one

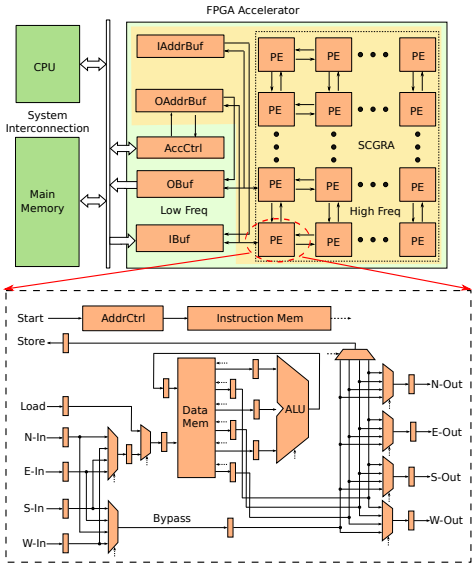


Fig. 2. SCGRA overlay based FPGA accelerator

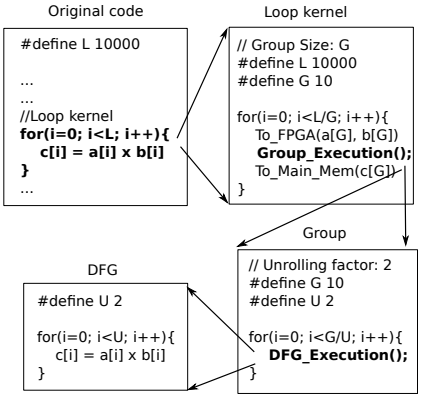


Fig. 3. Loop, group and DFG. The loop will be divided into groups. Each group will be partially unrolled and the unrolled part will be translated to DFG. IO transmission between FPGA and host CPU is performed in the granularity of a group.

hand, this technique is used to reduce the number of batching, which further helps to amortize the initial communication cost. On the other hand, it also results in larger on-chip memory overhead. The proposed customization framework can be used to make the right design choices to achieve an optimal design.

C. SCGRA overlay compilation

With pre-built SCGRA overlay library and customized overlay configuration, the corresponding FPGA accelerator can be generated rapidly, which is also the basis of the high-productivity loop accelerator design framework. Figure 4 presents the detailed SCGRA overlay compilation. With the specified loop unrolling and grouping factor, DFG is generated and scheduled to the SCGRA overlay of the accelerator. After the scheduling, control words are extracted, and they can further be integrated into the pre-built FPGA accelerator bitstream creating the final FPGA loop accelerator bitstream. The compilation process typically completes in a few seconds

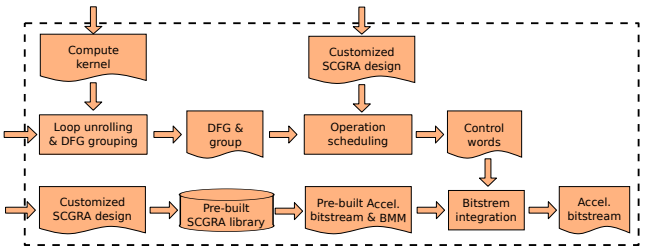


Fig. 4. Rapid SCGRA overlay compilation

as illustrated in [10] which is particularly important during early application development phases.

IV. SCGRA Overlay Based FPGA Accelerator Customization

Application-specific customization provides unique opportunity to reduce the resource consumption and improve performance of the resulting accelerators. However, taking the system as a black box and exhaustively searching all the possible configurations can be inefficient and slow. In this work, by taking advantage of the regularity of the SCGRA overlay based FPGA accelerator, we can reduce the complex customization problem to a much simpler sub design space exploration (DSE) together with a simplified search problem. With the customization, optimized application-specific nested loop accelerator can be produced efficiently.

A. Customization problem formulation

In this section, we will formalize the customization problem of the nested loop acceleration on an SCGRA overlay based FPGA accelerator. Various design constraints including energy consumption and hardware resource consumption can be used while hardware resource consumption is taken as an example here.

TABLE I
Design Parameters of Nested Loop Acceleration [1]

Design Parameters		Denotation
Nested Loop Compilation	Loop Unrolling Factor	$\boldsymbol{u} = (u_0, u_1, ...)$
	Grouping Factor	$\boldsymbol{g} = (g_0, g_1, ...)$
Overlay Configuration	SCGRA Topology	2D Torus, fixed
	SCGRA Size	$r \times c$
	Data Width	W_0
	Data Mem	$D_0 \times W_0$
	Input Buffer	$D_1 \times W_0$
	Output Buffer	$D_2 \times W_0$
	Instruction Mem	$D_3 \times W_1$
	Input Address Buffer	$D_4 \times W_2$
	Output Address Buffer	$D_5 \times W_3$
	Operation Set	fixed
	Implementation Frequency	f, fixed
	Pipeline Depth	fixed

Suppose $\boldsymbol{\Psi}$ represents the overall nested loop acceleration design space. $\boldsymbol{C} \in \boldsymbol{\Psi}$ represents a possible configuration in the design space and it includes a number of design parameters as listed in Table I. Assume that the loop to be accelerated has n nested levels and loop count can be denoted as

[1]The parameters are all customizable in the proposed design framework except the ones that are clearly identified as fixed.

$l = (l_1, l_2, ..., l_n)$. $R = (R_1, R_2, R_3, R_4)$ stands for the FPGA resource (i.e. BRAM, DSP, LUT and FF) that are available on a target FPGA and $ResConsumption(\boldsymbol{C}, i)$ denotes the four different types of FPGA resource consumption. $In(\boldsymbol{g})$ and $Out(\boldsymbol{g})$ stand for the amount of input and output of a group. Similarly, $In(\boldsymbol{u})$ and $Out(\boldsymbol{u})$ stand for the amount of input and output of a DFG. $DFGCompuTime(\boldsymbol{C})$ represents the number of cycles needed to complete the DFG computation. α_i and β_i are constant coefficients depending on target platform where $i = (1, 2, ...)$. With these denotations, the customization problem targeting minimum run time can be formulated as follows:

Minimize

$$RunTime(\boldsymbol{C}) = CompuTime(\boldsymbol{C}) + CommuTime(\boldsymbol{C}) \tag{1}$$

subject to

$$\begin{aligned} &ResConsumption(\boldsymbol{C}, i) \leq R_i, i = 1, 2, 3, 4 \\ &In(g) \leq D_1 \\ &Out(g) \leq D_2 \\ &DFGCompuTime(\boldsymbol{C}) \leq D_3 \\ &\prod_{i=1}^{n} \frac{g_i}{u_i} \times In(u) \leq D_4 \\ &\prod_{i=1}^{n} \frac{g_i}{u_i} \times Out(u) \leq D_5 \end{aligned} \tag{2}$$

$RunTime(\boldsymbol{C})$ represents the number of cycles needed to compute the loop on the CPU-FPGA system. It consists of both the time consumed for computing on FPGA and communication between FPGA and host CPU, and it can be calculated using Equation 1.

Since the unrolled part of the loop will be translated to DFG and then scheduled to the SCGRA overlay. Thus the DFG computation time is essentially a function of $\mathbf{u}$, r and c, and it can also be denoted by $DFGCompuTime(\mathbf{u}, r, c)$. The nested loop is computed by repeating the same DFG execution, and the nested loop computation can be calculated using Equation 3.

$$CompuTime(\boldsymbol{C}) = \prod_{i=1}^{n} \frac{l_i}{u_i} \times DFGCompuTime(\mathbf{u}, r, c) \tag{3}$$

DMA is typically used for the bulk data transmission. Communication cost per data can be modeled with a piecewise linear function and thus DMA latency can be calculated using $DMA(x)$ where x represents the amount of DMA transmission. The communication time of the whole nested loop can be calculated by Equation 4.

$$CommuTime(\boldsymbol{C}) = \prod_{i=1}^{n} \frac{l_i}{g_i} \times (DMA(In(\mathbf{g})) + DMA(Out(\mathbf{g}))) \tag{4}$$

Hardware resource on FPGA mainly includes DSP, LUT, FF and BRAM (block RAM). LUT, FF and DSP consumption can be roughly estimated with a linear function of SCGRA size and can be calculated using Equation 5. BRAM consumption $ResConsumption(\boldsymbol{C}, 1)$ which is slightly different from LUT, FF and DSP consumption can be calculated precisely based on the memory block configurations.

$$ResConsumption(\boldsymbol{C}, i) = \alpha_i \times r \times c + \beta_i, (i = 2, 3, 4) \tag{5}$$

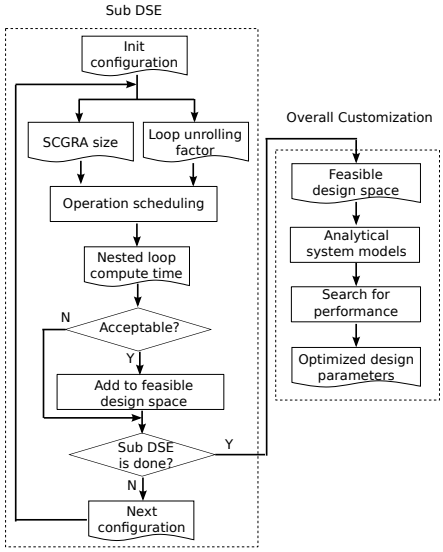


Fig. 5. System customization framework.

B. Customization framework

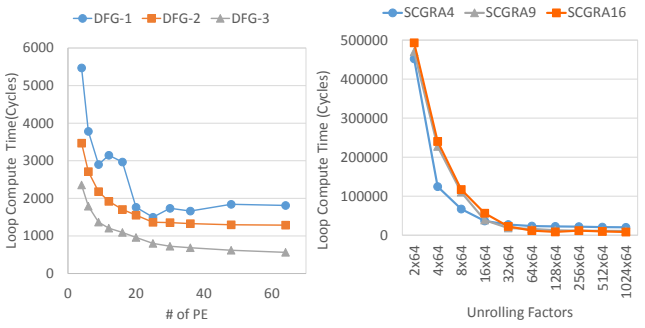
Figure 5 illustrates the overview of the customization framework. It can be roughly divided into two parts. In the first part, a sub DSE targeting loop execution time is performed and the feasible design space can be obtained. Since loop execution time is determined by the operation scheduling which simply depends on the loop unrolling factor and SCGRA size, the sub DSE is much simpler compared to the overall system DSE which includes more than 10 design parameters. In the second part, each configuration in the feasible design space will be evaluated. Instead of using simulation based methods, analytical models are employed to estimate the accelerator metrics such as performance and hardware resource consumption. These analytical models are accurate because of the regularity of the SCGRA overlay. Even though the feasible design space is still large, it is fast to evaluate all the configurations in it. After the evaluation process, customization for best performance becomes trivial and the customized design parameters can be obtained immediately.

Suppose Φ denotes the feasible design space. ϵ indicates the percentage of the performance benefit obtained by the increase of loop unrolling or SCGRA size. It is a user defined threshold and must be small enough to prune the configurations that are inappropriate. The configurations in Φ must satisfy Equation 6 and Equation 7.

$$\begin{aligned} &\forall \boldsymbol{C} = (..., \boldsymbol{u}, r, c, ...) \in \Phi, \boldsymbol{C'} = (..., \boldsymbol{u'}, r', c', ...) \in \Phi, \\ &(r + 1 == r' \text{ and } c == c') \text{ or } (r == r' \text{ and } c + 1 == c') : \\ &\frac{CompuTime(\boldsymbol{C}) - CompuTime(\boldsymbol{C'})}{CompuTime(\boldsymbol{C})} > \epsilon \end{aligned} \tag{6}$$

$$\begin{aligned} &\forall \boldsymbol{C} = (..., \boldsymbol{u}, r, c, ...) \in \Phi, \boldsymbol{C'} = (..., \boldsymbol{u'}, r, c, ...) \in \Phi, \\ &\boldsymbol{u} \text{ and } \boldsymbol{u'} \text{ are consecutive unrolling factors :} \\ &\frac{CompuTime(\boldsymbol{C}) - CompuTime(\boldsymbol{C'})}{CompuTime(\boldsymbol{C})} > \epsilon \end{aligned} \tag{7}$$

Each feasible configuration $\boldsymbol{C} \in \Phi$ must have gone through the scheduling and thus the corresponding scheduling result is



(a) (b)

Fig. 6. The design parameters typically have monotonic influence on the loop computation time and the computation time benefit degrades with the increase of the design parameter. (a) SCGRA Size, the SCGRA topology used are torus with 2×2, 3×2, 3×3, ... while DFG-1, DFG-2 and DFG-3 are DFGs extracted from matrix-matrix multiplication, fir and Kmean respectively. (b) Unrolling Factor, the loop used is a 63-tap Fir with 1024 input.

known. Consequently, the computation time of the loop kernel and minimum instruction memory depth are available as well. Then we can further evaluate the performance of each feasible configuration using the models built in previous section and obtain the optimized configuration through a simple search.

In addition, a series of experiments on Zedboard as shown in Figure 6 demonstrate that SCGRA size and unrolling factor present a clear monotonic influence on the loop compute time. The performance benefit of loop unrolling and increase of SCGRA size drops gradually. This observation further helps to simplify the sub DSE with a simple branch and bound algorithm.

V. Experiments and Results

In the experiments, we measured the time needed to customize the loop accelerators and compared the performance of the resulting accelerators to that of an hard ARM processor.

A. Experiment setup

The customization runtime was obtained using a computer with Intel(R) Core(TM) i5-3230M CPU and 8GB RAM. Zedboard which has an ARM processor and an FPGA was used as the computation system. PlanAhead 14.7 was used for the SCGRA overlay based design. The customized overlay implementations on Zedboard run at 250MHz. To perform the customization, ϵ is set to be 0.05 and all the resource on Zedboard is set to be the resource constraint. Software runtime is obtained from ARM processor of Zedboard.

In this work, we take four applications including Matrix Multiplication (MM), FIR, Kmean(KM) and Sobel Edge Detector (SE) as our benchmark. The configurations of the benchmark are detailed in Table II.

B. Customization time

Figure 7 shows the customization time of both the proposed two step (TS) customization and an exhaustive search based customization (ES). TS typically completes the customization in 10 minutes to 20 minutes and it is around 100x faster than the ES on average. In particular, ES is extremely slow on MM which has three levels of loop with relatively large loop count and thus larger design space. Though TS also needs longer time to complete the customization, it skips most of the unfeasible configurations and the runtime is less sensitive to the size of the design space.

TABLE II
Benchmark Configurations

Benchmark	Parameters	Loop Structure
MM	Matrix Size(100)	$100 \times 100 \times 100$
FIR	# of Input (10000) # of Taps+1 (50)	10000×50
SE	# of Vertical Pixels (128) # of Horizontal Pixels (128)	$128 \times 128 \times 3 \times 3$
KM	# of Nodes(5000) # of Centroids(4) # of Dimensions(2)	$5000 \times 4 \times 2$

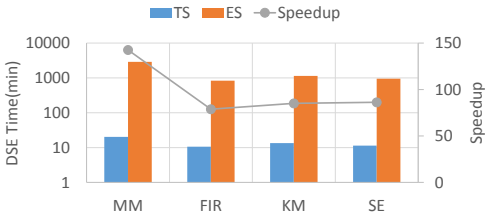


Fig. 7. Benchmark customization time using both TS and ES

TABLE III
Accelerator Configurations [2]

MM	Base	$(1 \times 2 \times 100, 4 \times 2 \times 100, 5 \times 5, 1k, 2k)$
	TS	$(1 \times 5 \times 100, 50 \times 5 \times 100, 4 \times 4, 1k, 8k)$
	ES	$(1 \times 5 \times 100, 25 \times 5 \times 100, 5 \times 4, 1k, 8k)$
FIR	Base	$(10 \times 50, 100 \times 50, 3 \times 3, 1k, 2k)$
	TS	$(50 \times 50, 2000 \times 50, 4 \times 4, 1k, 4k)$
	ES	$(100 \times 50, 5000 \times 50, 5 \times 4, 1k, 8k)$
SE	Base	$(4 \times 4 \times 3 \times 3, 128 \times 128 \times 3 \times 3, 3 \times 2, 1k, 8k)$
	TS	$(16 \times 16 \times 3 \times 3, 128 \times 128 \times 3 \times 3, 4 \times 4, 1k, 4k)$
	ES	$(16 \times 16 \times 3 \times 3, 128 \times 128 \times 3 \times 3, 5 \times 4, 1.5k, 4k)$
KM	Base	$(25 \times 4 \times 2, 2500 \times 4 \times 2, 4 \times 3, 1k, 8k)$
	TS	$(125 \times 4 \times 2, 625 \times 4 \times 2, 5 \times 5, 1k, 2k)$
	ES	$(125 \times 4 \times 2, 625 \times 4 \times 2, 5 \times 5, 1k, 2k)$

C. Customized accelerator performance

In order to demonstrate the quality of proposed framework, we compared the performance of the accelerators with a random configuration as well as customized configurations obtained using both TS and ES. The detailed configurations of the accelerators are listed in Table III. The performance comparison is shown in Figure 8. It can be found that the customized accelerators obtained using TS achieve quite close performance to the ones customized through ES. Particularly, the customized accelerator achieves up to 10X speedup over the ARM processor on the benchmark. For FIR, SE and KM, the speedup is promising. MM has relatively low compute-IO rate and the single input and output between the on-chip buffer and the SCGRA overlay limits the performance of the accelerator. This problem can hopefully be alleviated by appropriate on-chip buffer partition, which will be supported in the proposed framework in future.

[2]The configurations include loop unrolling factor, grouping factor, SCGRA array size, instruction memory depth and IO buffer depth

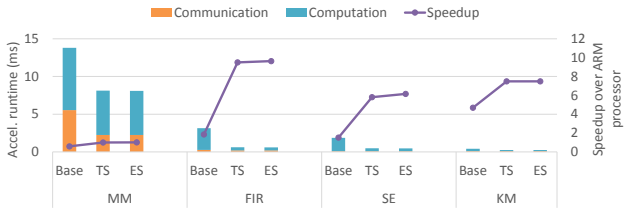


Fig. 8. Customized FPGA loop accelerator performance

VI. Conclusion

In this work, we have presented an automatic nested loop acceleration framework that is based on a soft coarse-grained reconfigurable array overlay. We have demonstrated that by taking advantage of the regularity of the overlay, intensive system customization specific to the given user application can be performed efficiently, resulting in up to 5 times performance improvement over solutions without customization at the cost of 10 to 20 minutes additional tools run time. Overall, the framework is able to generate accelerators that achieve up to 10 times speed up over software running on the host processor, resulting in a high design productivity experience for software programmers.

Acknowledgment

This work was supported in part by the Research Grants Council of Hong Kong project ECS 720012E and the Croucher Innovation Award 2013.

References

[1] E. Chung *et al.*, "Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs?" in *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, Dec 2010, pp. 225–236.

[2] U. Bondhugula, J. Ramanujam, and P. Sadayappan, "Automatic mapping of nested loops to FPGAs," in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007, San Jose, California, USA, March 14-17, 2007*, 2007, pp. 101–111. [Online]. Available: http://doi.acm.org/10.1145/1229428.1229446

[3] J. Cong *et al.*, "High-level synthesis for FPGAs: From prototyping to deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 473–491, 2011.

[4] D. Grant, C. Wang, and G. G. Lemieux, "A CAD framework for Malibu: An FPGA with time-multiplexed coarse-grained elements," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 123–132. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950441

[5] J. Coole and G. Stitt, "Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing," in *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, Oct 2010, pp. 13–22.

[6] A. Brant and G. Lemieux, "ZUMA: An open FPGA overlay architecture," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, April 2012, pp. 93–96.

[7] D. Capalija and T. Abdelrahman, "A high-performance overlay architecture for pipelined execution of data flow graphs," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, Sept 2013, pp. 1–8.

[8] R. Ferreira *et al.*, "An FPGA-based heterogeneous coarse-grained dynamically reconfigurable architecture," in *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*. ACM, 2011, pp. 195–204.

[9] D. Kissler *et al.*, "A dynamically reconfigurable weakly programmable processor array architecture template." in *ReCoSoC*, 2006, pp. 31–37.

[10] C. Liu, C. Yu, and H.-H. So, "A soft coarse-grained reconfigurable array based high-level synthesis methodology: Promoting design productivity and exploring extreme FPGA frequency," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, April 2013, pp. 228–228.

[11] A. K. Jain, S. A. Fahmy, and D. L. Maskell, "Efficient overlay architecture based on DSP blocks," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, May 2015, pp. 25–28.

[12] A. Al-Dujaili *et al.*, "Guppy: A GPU-like soft-core processor," in *Field-Programmable Technology (FPT), 2012 International Conference on*, Dec 2012, pp. 57–60.

[13] P. Yiannacouras, J. G. Steffan, and J. Rose, "Fine-grain performance scaling of soft vector processors," in *Proceedings of the 2009 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, ser. CASES '09. New York, NY, USA: ACM, 2009, pp. 97–106. [Online]. Available: http://doi.acm.org/10.1145/1629395.1629411

[14] A. Severance and G. Lemieux, "Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor," in *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*, Sept 2013, pp. 1–10.

[15] D. Unnikrishnan, J. Zhao, and R. Tessier, "Application specific customization and scalability of soft multiprocessors," in *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*, April 2009, pp. 123–130.

[16] I. Lebedev *et al.*, "MARC: A many-core approach to reconfigurable computing," in *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, Dec 2010, pp. 7–12.

[17] P. Yiannacouras, J. Steffan, and J. Rose, "Exploration and customization of FPGA-based soft processors," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, no. 2, pp. 266–277, Feb 2007.

[18] D. Capalija and T. Abdelrahman, "An architecture for exploiting coarse-grain parallelism on FPGAs," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, Dec 2009, pp. 285–291.

[19] C. E. LaForest and J. G. Steffan, "OCTAVO: An FPGA-centric processor family," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: ACM, 2012, pp. 219–228. [Online]. Available: http://doi.acm.org/10.1145/2145694.2145731

[20] H. Y. Cheah, S. Fahmy, and D. Maskell, "iDEA: A DSP block based FPGA soft processor," in *Field-Programmable Technology (FPT), 2012 International Conference on*, Dec 2012, pp. 151–158.

[21] R. Tessier and W. Burleson, "Reconfigurable computing for digital signal processing: A survey," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 28, no. 1-2, pp. 7–27, 2001.

[22] K. Compton and S. Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Computing Surveys (csuR)*, vol. 34, no. 2, pp. 171–210, 2002.

[23] M. X. Yue, D. Koch, and G. G. Lemieux, "Rapid overlay builder for Xilinx FPGAs," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, May 2015, pp. 17–20.

[24] K. Compton and S. Hauck, "Totem: Custom reconfigurable array generation," in *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, March 2001, pp. 111–119.

[25] L. Zhou *et al.*, "Application-specific coarse-grained reconfigurable array: architecture and design methodology," *International Journal of Electronics*, no. ahead-of-print, pp. 1–14, 2014.

[26] N. R. Miniskar *et al.*, "Retargetable automatic generation of compound instructions for CGRA based reconfigurable processor applications," in *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2014 International Conference on*. IEEE, 2014, pp. 1–9.

[27] J. Coole and G. Stitt, "Adjustable-cost overlays for runtime compilation," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, May 2015, pp. 21–24.

[28] C. Y. Lin and H. K.-H. So, "Energy-efficient dataflow computations on FPGAs using application-specific coarse-grain architecture synthesis," *SIGARCH Comput. Archit. News*, vol. 40, no. 5, pp. 58–63, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2460216.2460227

ThreadPoolComposer – An Open-Source FPGA Toolchain for Software Developers

Jens Korinth, David de la Chevallerie, Andreas Koch
Embedded Systems and Applications Group (ESA)
Technische Universität Darmstadt
Darmstadt, Germany
Email: {jk, dc, ahk}@esa.cs.tu-darmstadt.de

***Abstract*—**
This extended abstract presents ThreadPoolComposer, a high-level synthesis-based development framework and meta-toolchain that provides a uniform programming interface for FPGAs portable across multiple platforms.

***Index Terms*—FPGA; application programming interface; design automation; accelerators; higher level synthesis**

I. Introduction

In recent years, major advances in *High-Level Synthesis (HLS)* have spawned a new generation of hardware compilers (such as LegUp [1] or Nymble [2] in the academic domain, or Xilinx Vivado HLS in industry) which can generate efficient, behaviorally equivalent hardware for computing kernels described in C/C++. Until recently, these tools were burdened not only with tackling the highly complex task of generating hardware from a C/C++ specification, but also with the equally daunting task of system synthesis, namely providing an entire hardware/software environment for the generated hardware kernels. This encompasses, e.g., defining and connecting to memories, managing host/FPGA communication and making the FPGA accessible using appropriate software interfaces.

ThreadPoolComposer aims to divide the task of *generating an FPGA hardware design* into the actual **HLS problem**, and the **problem of generating on-chip micro-architectures at the system level**. The main goals of ThreadPoolComposer are to provide *an easily customizable open-source tool suitable for researchers* investigating the latter problem, and *a common benchmark environment for researchers* working on HLS tools, while isolating software developers from the low-level mechanisms.

ThreadPoolComposer was developed in context of the EU FP7 research project REPARA [3], which aims for an automated front-to-back development flow for heterogeneous parallel computers encompassing one or more of multi-core, GPU, FPGA, and DSP-based processing elements. The flow can begin with legacy C++ code which is then incrementally refactored into modern C++11/14, from which in turn high-level code suitable for the different target processors can be deduced.

II. ThreadPoolComposer

In the following, the ThreadPoolComposer toolchain and framework will be presented in a top-down approach, i.e., from the software interface down to the hardware bitstream generation for an FPGA device.

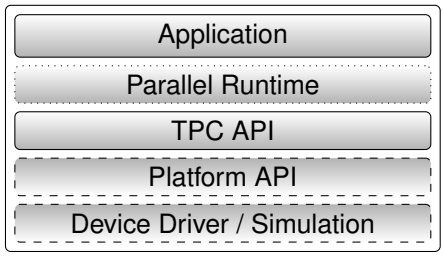


Fig. 1: ThreadPoolComposer Software Stack

A. TPC API

The TPC API is the upper-most API layer (see Fig. 1); either the application directly uses TPC API, or it uses a parallel runtime framework (such as OpenCL, FastFlow [4]) which interfaces with TPC API. Its core tasks are 1) device enumeration and management 2) data transfer to and from the device 3) job preparation and launching. Listing 1 shows an example snippet of a job launch:

```
/* allocate 1 KB on device */
tpc_handle_t h = tpc_device_alloc(dev, 1024);
/* copy array 'data' to device */
tpc_device_copy_to(dev, data, h, 1024, TPC_BLOCKING_MODE);
/* prepare a new job for kernel id #10 (magic) */
tpc_job_id_t j_id = tpc_device_acquire_job_id(dev, 10);
/* set argument #0 to handle h */
tpc_device_job_set_arg(dev, j_id, 0, sizeof(h), &h);
/* launch job */
tpc_device_job_launch(dev, j_id, TPC_BLOCKING_MODE);
/* call blocks until completed, so get return value */
int r = 0;
tpc_device_job_get_return(dev, j_id, sizeof(r), &r);
printf("result of job: %d\n", r);
/* release job id */
tpc_device_release_job_id(dev, j_id);
/* release device memory */
tpc_device_free(dev, h);
```

Listing 1: TPC API Example

First, a small block of memory is allocated on the device via `tpc_device_alloc`, to which some data is copied via `tpc_device_copy_to`. Then, a job is requested and prepared by setting the first argument to the memory handle, i.e., the kernel shall work with the data that has just been transferred to the device. A job can be launched on the device either in *blocking*

2nd International Workshop on *FPGAs for Software Programmers (FSP 2015)*, London, United Kingdom, September 1, 2015.

or non-blocking mode, i.e., the call returns after the job has finished, or immediately. Finally, the results are collected and the device memory is freed. This style is reminiscent of OpenCL, which was a deliberate choice to flatten the learning curve. Also note that we deliberately decided for manual data transfer management in order to give runtime schedulers optimization opportunities, e.g., by keeping intermediate results on the device between job executions. Such capabilities are currently being integrated into the FastFlow [4] run-time for heterogeneous parallel computers.

B. Platform API

A wide range of FPGA-based processing platforms exists, ranging from reconfigurable systems-on-chip to larger PCI Express-based accelerators. Each device is usually aimed at a very specific audience and designed with certain applications in mind, which benefit hugely from the chosen architecture. This diversity cannot be easily unified without giving up a significant amount of the appeal of FPGA platforms. Therefore, the Platform API is inserted as a secondary software abstraction layer beneath TPC API; its purpose is to implement all *device-specific functionality*, currently: 1) device memory management 2) access to hardware registers, device memory 3) device-host communication and feedback. Both APIs can be implemented as *shared libraries*, giving the additional benefit of being *exchangeable at runtime*. From the software developer's perspective this allows moving between platforms **without recompilation of the application**. This facilitates *design space exploration* for any given application and increases re-usability.

C. Compilation Toolchain

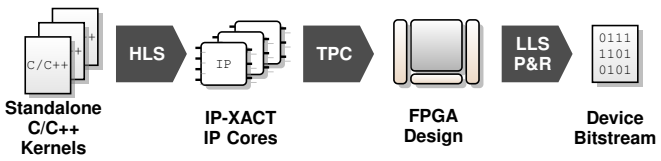


Fig. 2: TPC Compilation Flow

The overall compilation flow with ThreadPoolComposer is depicted in Fig. 2: Standalone C/C++ kernels have been extracted from the application and behaviorally equivalent IP cores are generated using *Xilinx Vivado HLS*. ThreadPoolComposer instantiates and arranges IP cores according to the given Composition, first creating a ThreadPool micro-architecture, which is wrapped in a Platform to yield a complete, synthesizable design. Finally, *Low-Level Synthesis (LLS)* is performed using the FPGA vendor toolchain (Xilinx Vivado).

The central idea behind ThreadPoolComposer is to *use HLS only as a C-to-Hardware compiler* at the level of individual accelerators, as opposed to being used as a *C-to-System compiler*, which would need to create an entire hardware system of accelerators, as well as internal and external interfaces, etc. While this is possible, it is rather awkward. Instead, the

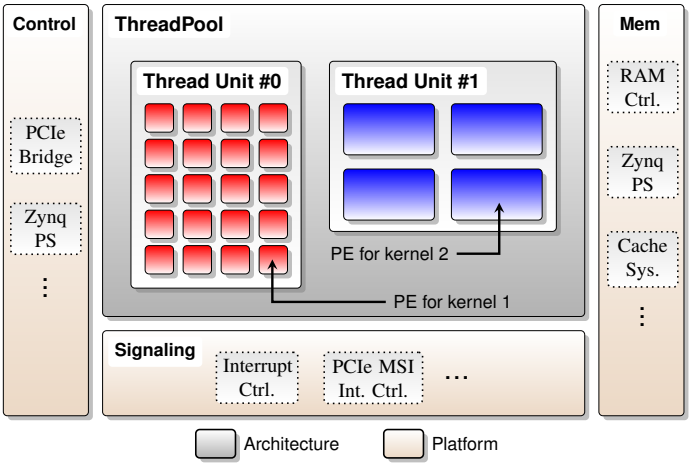


Fig. 3: FPGA Design Organization

developer identifies and extracts computational kernels from the application (probably using tool support), then selects a Platform, i.e., a device or device family of FPGAs, and specifies a Composition to the ThreadPoolComposer toolchain: Such a Composition defines 1) the kernels to be used in the design 2) the desired number of parallel processing elements (PEs) for each kernel (i.e., the degree of parallelism for each kernel) and 3) an Architecture, i.e., a construction template for the organization of the PEs in the design. Fig. 3 illustrates the general organization of the design: The Architecture defines the template to instantiate an on-chip organization of PEs called the ThreadPool, which connects to the host and memory via an hardware infrastructure instantiated by a template provided by the chosen Platform. Note that the dependencies between the template types have been minimized to enable maximal re-use of existing Architectures on new Platforms. This also facilitates comparisons, e.g., of different Architectures on a given Platform. The toolchain is based on *Scala/SBT* and the structural templates are written in *Vivado IP Integrator Tcl*, which makes ThreadPoolComposer very easy to customize, modify, and extend.

III. Platform Evaluation

ThreadPoolComposer is a work-in-progress and has not undergone thorough optimization yet. Currently, the system supports three Platforms using different classes of FPGA evaluation boards: The *zedboard* features a Zynq-7000 series XC7Z020-CLG484-1 FPGA with F_{max} of 100 MHz and a dual-core ARM Cortex A9 at 666 MHz as host processor running Xilinx Linux 3.17.0. Xilinx' ZC706 is a larger version of the same system, using a XC7Z054-FFG900-2 FPGA with F_{max} of 250 MHz and the same Cortex A9 running at 800 Mhz. Finally, the VC709 uses a 8x PCIe Gen3 interface based on **ffLink** [5] on a host with an eight-core Intel Xeon E5-1620v2 running at 3.7 GHz and Linux 3.19.5.

Fig. 4 shows the average data throughput in an otherwise idle system: Obviously, the VC709 benefits hugely from its PCIe interface, which transfers up to ≈4.2GiB/s at a chunk

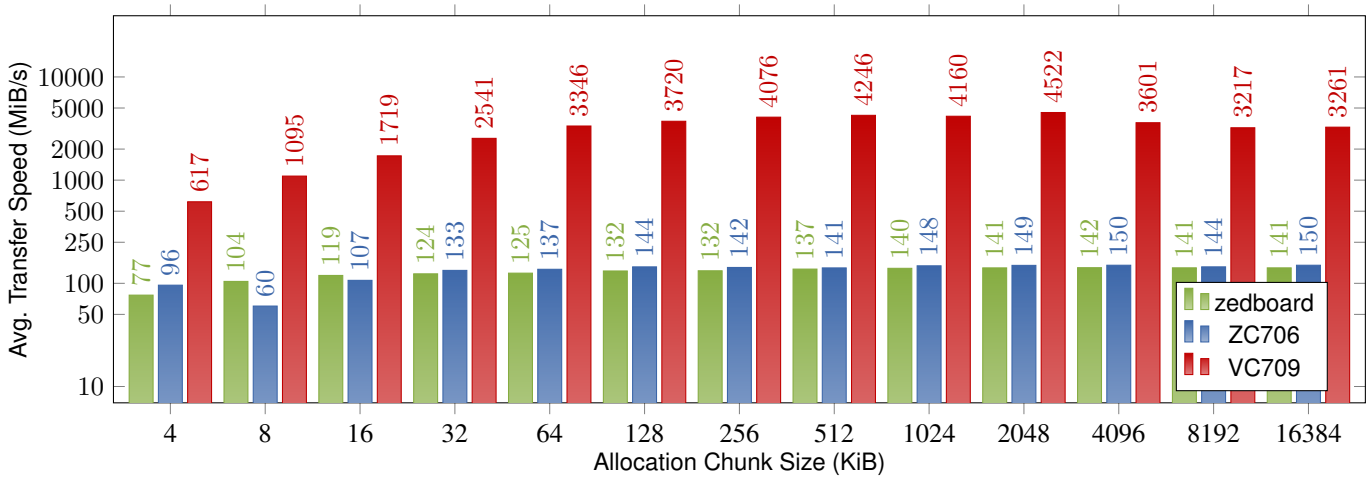


Fig. 4: Average bidirectional transfer rates between FPGA and host (i.e., user application memory) in MiB/s.

size of 512 KiB (and even more for much larger chunks, see [5]). The Zynq Platforms currently use kernel DMA buffers for the transfers, and their allocation leads to a significant slowdown. A *zero-copy approach* is currently under development to address this deficiency.

Fig. 5 depicts the interrupt latencies of the three platforms: To evaluate hardware/software round-trip time, we used a hardware counter to count the clock cycles between raising an interrupt (in hardware) and receiving the acknowledgement from the software (also in hardware). This measurement includes all intermediate software layers from OS level up to user application level. Latencies range from 3.2μs up to 22.5μs; shortest latencies can be achieved at the shortest kernel runtimes $\leq 10\mu s$ (calling thread is not put to sleep at all). Surprisingly, even though the VC709 has to transport interrupts via PCIe packets (and not dedicated wires), the latencies are significantly lower (almost by 2x). This is primarily due to the eight-core Xeon E5 running at 5x the speed of the ARMs on the zedboard.

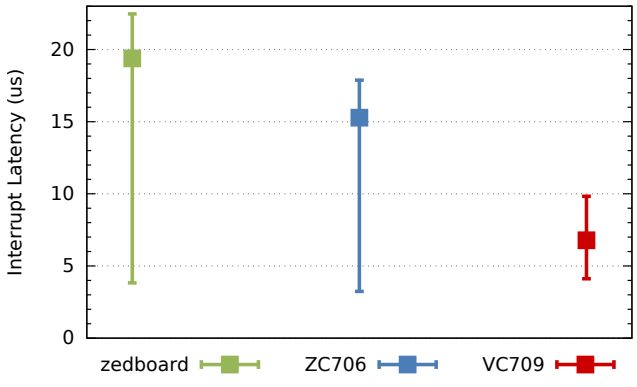


Fig. 5: Interrupt latencies: measured round-trip time (max/avg/min) between a hardware kernel signaling an interrupt and then receiving an acknowledgement from the host.

IV. Conclusion & Future Work

ThreadPoolComposer is an open-source meta toolchain which facilitates the exploration on-chip microarchitectures for FPGA accelerators, comparison of HLS tools and separates HLS from system-on-chip architecture generation. It furthermore provides a unified API for software developers, which can be used with every combination of ThreadPoolComposer Platforms and Architectures, thus improving the separation of concerns, and provides a solid basis for future automated architecture exploration efforts. ThreadPoolComposer currently supports the Zynq and zedboard devices, as well as the Xilinx VC709 with PCIe Gen3 x8 support. In future work, we aim to further increase the performance of the hardware designs by developing custom IP cores and integrate them by on-the-fly hardware generation via the Chisel language [6].

ThreadPoolComposer will be released in open-source form from the Downloads *section of www.esa.cs.tu-darmstadt.de later this year.*

References

[1] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. S. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An Open Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems," *Embedded Computing Systems (TECS), ACM Transactions on*, 2013.

[2] J. Huthmann, B. Liebig, and A. Koch, "Hardware/software co-compilation with the Nymble system," *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 8th Int. Workshop on*, 2013.

[3] "REPARA - Reengineering and Enabling Performance and poweR of Applications," 2013. [Online]. Available: http://www.repara-project.eu

[4] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati, "Fastflow: high-level and efficient streaming on multi-core," in *Programming Multi-core and Many-core Computing Systems*, ser. Parallel and Distributed Computing, S. Pllana and F. Xhafa, Eds. Wiley, 2014.

[5] D. de la Chevallerie, J. Korinth, and A. Koch, "ffLink: A Lightweight High-Performance Open-Source PCI Express Gen3 Interface for Reconfigurable Accelerators," *Highly Efficient Accelerators and Reconfigurable Technologies (HEART), International Symposium on*, 2015.

[6] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: constructing hardware in a scala embedded language," *Design Automation Conference (DAC), 49th ACM/EDAC/IEEE*, 2012.

Framework for Application Mapping over Packet-switched Network of FPGAs : Case studies

Vinay B. Y. Kumar*, Pinalkumar Engineer*, Mandar Datar*,
Yatish Turakhia†, Saurabh Agarwal*, Sanket Diwale‡ and Sachin B. Patkar*

Department of Electrical Engineering, Indian Institute of Technology Bombay, Mumbai, India
Email:*{vinayby, pje, mandardatar, saurabh, patkar}@ee.iitb.ac.in, †yatisht@standford.edu, ‡sanket.diwale@epfl.ch

***Abstract*—The algorithm-to-hardware High-level synthesis (HLS) tools today are purported to produce hardware comparable in quality to handcrafted designs, particularly with user directive driven or domains specific HLS. However, HLS tools are not readily equipped for when an application/algorithm needs to scale. We present a (work-in-progress) semi-automated framework to map applications over a packet-switched network of modules (single FPGA) and then to seamlessly partition such a network over multiple FPGAs over quasi-serial links. We illustrate the framework through three application case studies: LDPC Decoding, Particle Filter based Object Tracking, and Matrix Vector Multiplication over GF(2). Starting with high-level representations of each case application, we first express them in an intermediate message passing formulation, a model of communicating processing elements. Once the processing elements are identified, these are either handcrafted or realized using HLS. The rest of the flow is automated where the processing elements are plugged on to a configurable network-on-chip (CONNECT) topology of choice, followed by partitioning the 'on-chip' links to work seamlessly across chips/FPGAs.**

I. Introduction

As applications targeting FPGAs grow more pervasive or when they need to scale, there are matching demands on logic capacity as well as resources such as special-function on-chip resources, I/O and reliable multi-gigabit transceivers. Moore scaling enabled meeting these demands in large part. As with general purpose processors, more than Moore scaling with FPGAs is enabled by multiple FPGA platforms—the classic use-cases of which are ASIC prototyping, Emulation and Hardware-acceleration of applications and also more recently for datacenter applications [1].

Although commercial HLS tools such as Vivado [2]—given good user directives—are capable of producing hardware of quality comparable to handcrafted designs, it is not within the ready scope of HLS tools to address the issue of scalability. This problem becomes even more tricky because of the fragmentation in the ways the multi-FPGA platforms are built, particularly in terms of the variety in the nature of host to FPGA/s and inter FPGA links, and underlying custom interfaces. Dally et. al. [3] recently advocated for design productivity through modular designs with standardized interfaces on a network-on-chip abstraction. In the current context, such a standard interface can abstract the variety in the physical links.

In this work we begin to explore the scalability of applications/algorithms (used interchangeably henceforth)—particularly those amenable to be expressed in a data-flow manner—through a network abstraction, and an automation framework that would simplify exploration of this complex design space in mapping to a given multi-FPGA platform. In particular, we map the application task graph to a packet-switched Network-on-Chip (NoC), and extend the NoC abstraction across FPGAs communicating over quasi-serial links. The path from a higher-level specification of the application to a task-graph with precedence constraints, followed by coarsening and identifying the partition across chips is not discussed in this work (related earlier work: [4]).

We illustrate the framework through three cases studies that could use scalability, each of a different flavor—I. LDPC decoding, min-sum algorithm; II. Particle Filter based Object Tracking; and III. Matrix Vector Multiplication over GF(2). Case I naturally has a message passing structure, unlike II. For case III, although a more straightforward message passing model could have been used, as a way to highlight the role of a domain expert in this step, we use a novel sub-quadratic algorithm by Ryan Williams [5], this incidentally being its first hardware realization. For each case study, in phase-1, we

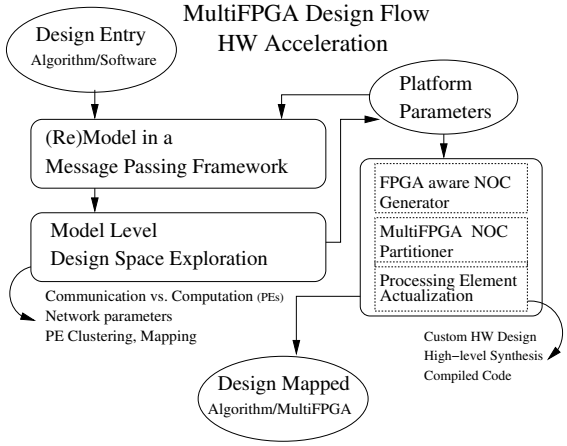


Fig. 1: Design flow: Scaling Hardware Acceleration

start with a high-level description of the algorithm, express it in message passing formulation, followed by realization of the processing elements either by HLS (Vivado) or custom design. Phase-2 automates the process of integrating these processing

2nd International Workshop on *FPGAs for Software Programmers (FSP 2015)*, London, United Kingdom, September 1, 2015.

elements onto a network-on-chip (NoC) architecture (auto-generated by a NoC Generator, CONNECT [6]), followed by seamlessly (in a manner oblivious to the designer) partitioning the NoC over multiple FPGAs where the NoC links crossing FPGAs are replaced by stitching-in quasi-serial links implemented over FPGA pins. In other words, this work flow expects the algorithm domain expert (software) to help express the original algorithm in a message passing model (phase-1), the rest of the flow is an automation that gives a scaled design over an NoC or multiple FPGAs. Figure 1 outlines the design flow.

This semi-automated framework is a work in progress, and was done with a little manual intervention for the case studies discussed.

A. Organization

The rest of the paper is organized as follows. Section II discusses phase-1 of the automation where the algorithm is expressed in a way that helps identification and synthesis of processing elements, followed by wrapping them with suitable adapters before plugging them to CONNECT NoC. Section III, phase-2 of the automation, describes the design of quasi-SERDES endpoints and the automation of partition of the NoC across multiple FPGAs. The next three sections IV, V, and VI discuss the specific case studies mentioned above.

II. Phase-1: Application mapping to NoC

A. Message passing modeling of the Application

The algorithm should first be expressed in a message passing formulation. This modeling, at the software level, is best done by the domain expert. The result is a model of software threads—corresponding to processing elements in hardware—communicating in a message passing fashion. For simplicity, we assume the body of the function/thread is executed after all the argument messages on received.

1) Note on compiler-driven automation: This phase too can be automated as long as the domain expert annotates the input high-level description appropriately. We have a compiler-driven toy automation flow (Figure 2) for this task, that partitions the Dataflow-Graph (DFG) extracted from a high-level description (straight line code) to be executed on a network of MIPS processors. The DFG parts are compiled to a minimal MIPS instruction set with network-push/pull instructions (FIFO-semantics) added to account for the communication between the DFG parts, taking into account the precedence constraints/schedule. [4] is a follow-up work in this direction focusing on fast scheduling and mapping.

B. Processing Element Realization and Interfacing to NoC

The hardware modules corresponding to the nodes of the message-passing graph identified in the previous step could either be designed by hand or a HLS tool. However, at this stage these modules are not yet network/NoC aware. Figure 3 shows the structure of a processing element that makes it pluggable on to an NoC. It consists of three modules: *Data collector*, *Data processing* and *Data distributor*. The *Data processing* module is the basic processing element that is synthesized out of the processes/functions from the previous step.

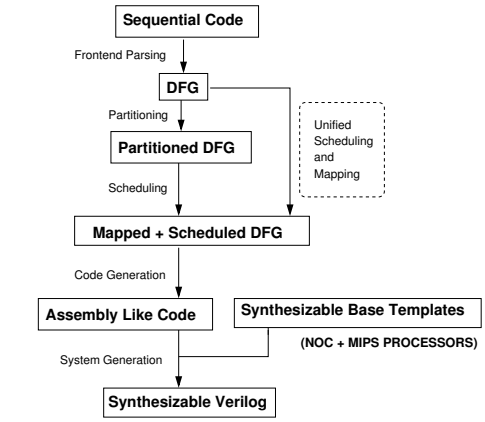


Fig. 2: Basic application partitioning and mapping tool flow

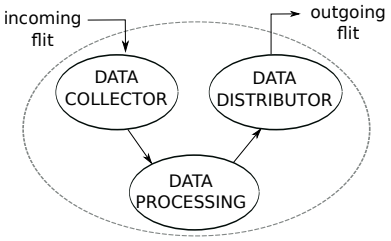
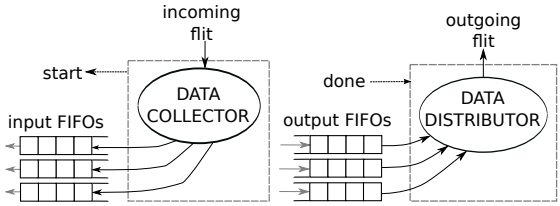
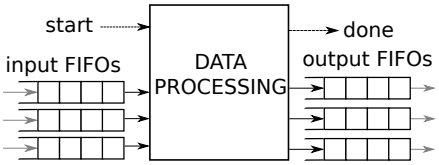


Fig. 3: Structure of Processing elements connecting to NoC



(a) *Data Collector* module (b) *Data distributor* module



(c) Computing elements for data processing

Fig. 4

Data collector and *Data distributor* modules—interfacing an NoC router on one side—are responsible for enabling external communication for processing elements over the NoC. Incoming data (in terms of Flits: basic units of data on NoC links) to the processing element is accepted at the router and processed by the *Data Collector* module, even with the flits arriving in an out-of-order fashion, and is put in appropriate

FIFOs corresponding to the input arguments of the processing element, the *Data processor*. Internal structure of *Data Collector* module is shown in Figure 4a. Once all the data is received and written into FIFOs, *start* is asserted to *Data processing* module. The interface of the *Data processing* module should be as in Figure 4c. Here, as the *start* is asserted, the input data is read from the *input FIFOs*, and once the computation is complete, the results are stored into *output FIFOs* and *done* is asserted. *Data distributor* module, as shown in Figure 4b, prepares the flit data (packet) from results and sends it to network interface of NoC router.

1) Automation: As mentioned earlier, the basic processing module could be designed using Verilog HDL or HLS. A script then generates a wrapper around such processing module in form of *Data collector* and *Data distributor* modules. Storage requirements of both input and output memory modules should be known a priori.

III. Phase-2: Partitioning NoC across multiple FPGAs

We use a freely available web-based synthesizable RTL generator for the Network-on-Chip (NoC) infrastructure, named CONNECT (Configurable Network Creation Tool). CONNECT [6] can be used for generating NoCs of arbitrary topology and supports a large variety of router and network configurations. Also, CONNECT incorporates a number of useful features fine-tuned for the FPGA platform.

In extending the NoC links across FPGAs, we require asynchronous links. However, the limited number of pins per FPGA would not support the typical router port-widths and radix counts. We therefore use serializer/deserializer (SERDES) blocks at the interfaces. One would typically use the dedicated multi-gigabit transceiver resources on the FPGA for SERDES links, but for this work, we develop a generic interfacing module that uses the GPIO pins available on any FPGA. As we use more than 1-pin to serialize the flit-transactions across a link (depending on the radix of the router, and the number of pins available), we call them quasi-SERDES.

Assuming an 8-wire physical link, these quasi-SERDES modules (on either side of a link) implement the following protocol—whenever a valid data (valid bit in the flit) in presented as input from router keep it in buffer and start sending 8 bits at a time with MSB first; similarly, whenever a valid 8 bit MSB is received reconstruct output data and put the data on the output port to the router.

Figure 5 shows an example partition of an NoC with four routers on two FPGAs. The router R0 (along with its processing element N0) is mapped onto a separate FPGA. Communication between FPGAs takes place using serializer/deserializer (quasi-SER/DES) links. The processsing elements $N1, 2, 3, 4$ here are as constructed earlier.

A. Automation

Given an NoC topology and an application mapped to it (as described above), and the decisions (presently user specified) as to 'cuts' that specify a partition on the NoC, an python script automates the process of generating required number of independent parts of the NoC and inserting a pair of quasi-SERDES endpoints on each NoC link cut. The independent part modules of the NoC are synthesized separately and programmed on respective FPGA boards. We have tested this framework between two Altera DE0-Nano boards, as well as two Xilinx Zynq Zedboads (ARM+FPGA).

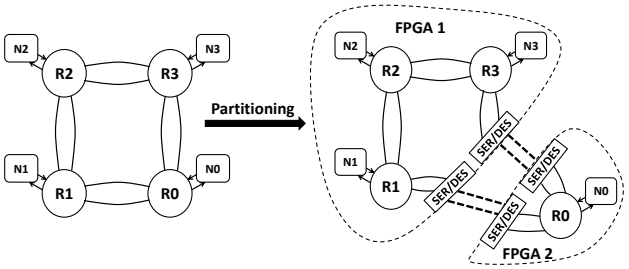


Fig. 5: Example 2-FPGA partition of an NoC

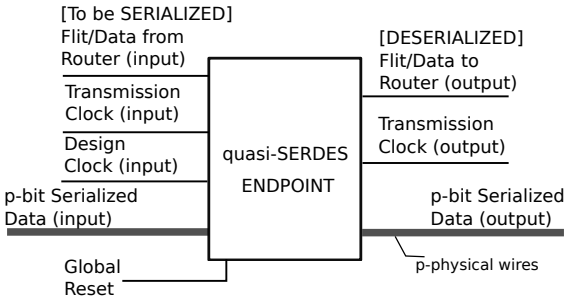


Fig. 6: Quasi-SER/DES Link Endpoint

IV. Case Study: LDPC Decoding

Listing 1 shows an outline of LDPC decoding based on the popular Min-sum algorithm. Number of data bits, to be decoded is N and $Niters$ is maximum number of iterations for LDPC decoding. Input to LDPC decoder is initial Log-Likelihood Ratio (LLR) of the data. LDPC decoding is done through Check nodes and Bit nodes iteratively, by passing message through dedicated channels between the nodes. Number of channels and interconnection between nodes depends on type of LDPC code. Here, we are using finite projective geometry based LDPC code [7][8] in $GF(2, 2^s)$ with $s = 1$. The message passing model is evident for this application and the processing nodes (the bit and check nodes) are also readily identified.

Listing 1: Outline of min-sum LDPC decoding

```
decoded[N] = minsum (data[N], Niter) {
  do {
    for (i = 0; i < N; i++) {
      // Initial LLR values
      u0(i) = data (i);
      uij = initial LLRs sent to Check node
      // j is degree of LDPC nodes

      //Check node processing
      vij = minimum(uij);
      //Bit node processing
      [uij, sum] = sum(vij);
    }
  } while(iterations<Niter);

 decoded[N] = sign(sum);
}
```

Code listing of check node processing and bit node processing shown in Listing 2 and 3 respectively.

Listing 2: Check node processing

```
[v1, v2, v3] = minimum (u1, u2, u3) {
        v1 = min(u2, u3);
        v2 = min(u1, u3);
        v3 = min(u1, u2);
}
```

Listing 3: Bit node processing

```
[sum, u1, u2, u3] = summation (u0, v1, v2, v3) {
        sum = u0 + v1 + v2 + v3;
        u1 = sum - v1;
        u2 = sum - v2;
        u3 = sum - v3;
}
```

Figures 7 and 8 show typical computing elements for check node and bit node processing respectively.

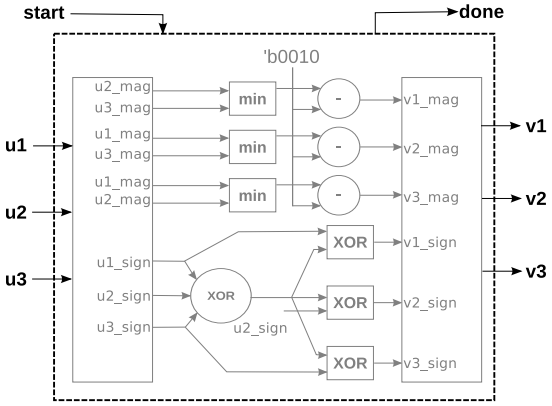


Fig. 7: Check node processing module

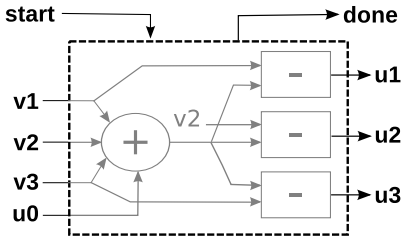


Fig. 8: Bit node processing module

Furthermore, these computing elements have been wrapped with *input FIFOs* and *output FIFOs* for interface compatibility with *Data Collector* and *Data distributor*. The wrappers were generated for both computing nodes, for interfacing them with CONNECT NoC. Table I shows resource utilization of bare computing nodes and computing nodes with wrapper.

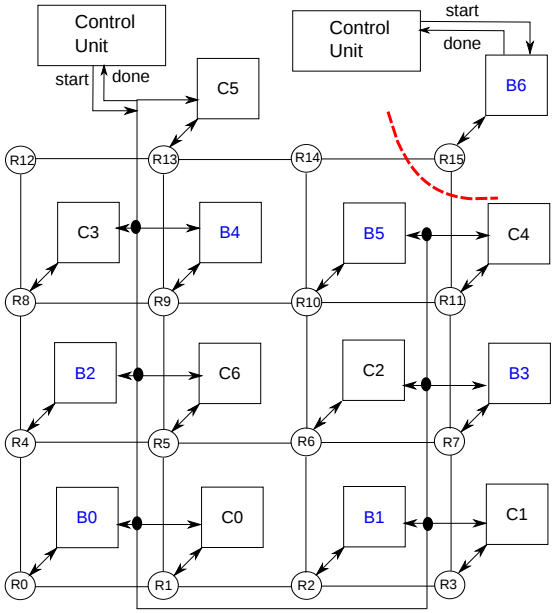


Fig. 9: LDPC decoder using 4×4 mesh CONNECT NoC

TABLE I: Resource utilization of computing nodes

Xilinx zc7020		Bit node		Check node	
		W/O wrapper	With wrapper	W/O wrapper	With wrapper
Resources	Available	Used	Used	Used	Used
Slice registers	106400	64	297	40	258
Slice LUTs	53200	110	261	73	199

For $N = 7$, both the wrapped computing nodes (bit node and check node), 7 each, are then interfaced to a 4×4 NoC as shown in Figure 9. Table II shows resource utilization of monolithic LDPC decoder (without NoC, same specs) and same with CONNECT NoC and wrapper. Resource utilization increases mainly due to the NoC being more generic than necessary. Dotted arc in Figure 9 indicates partitioning of NoC for multiple FPGA implementation.

TABLE II: Resource utilization of whole design

Xilinx zc7020		W/O wrapper		With NoC & wrapper	
Resources	Available	Used	%	Used	%
Slice registers	106400	866	1%	1429	1%
Slice LUTs	53200	1370	2%	1384	2%

V. Case Study: Particle Filter Based Object Tracking Algorithm

Important steps for our implementation [9] of object tracking based on Sequential Importance Sampling (SIS)

particle filter are listed below:

Particle Filtering based object tracking Algorithm:

- Calculate **reference histogram**
- For frames $k \rightarrow 2\ to\ n$
  - Initialize N samples $\{x_k^i\}_{i=1..N}$ (Gaussian distribution)
  - Distance weighted **candidate histograms** for N region of interest (ROI)
  - Calculate particle weights $\{w_k^i\}_{i=1..N}$ using **Bhattacharya distances** between reference histogram and candidate histograms
  - New center is estimated using **weighted mean calculation** using centers $\{x_k^i\}_{i=1..N}$ and weights $\{w_k^i\}_{i=1..N}$

Figure 10 shows implementation of particle filter based object tracking algorithm on NoC. For this, we have designed a standalone processing element to compute two important steps—the histogram calculation and calculation of Bhattacharya distances—of particle filter algorithm as shown in Figure 11. Figure 12 show the root node on Node-0 that orchestrates the computations on all other nodes.

Note that this is not necessarily the best way to map this application on an NoC, however, the approach makes exploring variations easier. For instance, the Bhattacharya coefficient calculation block within the current PE could be pulled out and shared as a resource over the network, as a separate processing element.

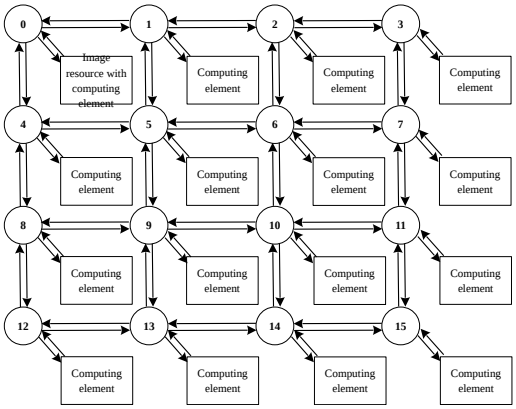


Fig. 10: The Particle filter processing elements mapped over NoC

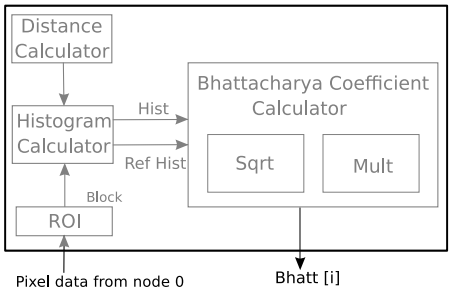


Fig. 11: Compute element for Particle filter based object tracking

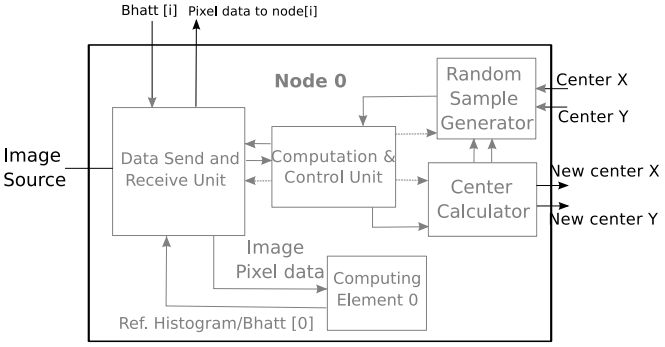


Fig. 12: The processing element on Node 0

Table III shows resource utilization of single processing element without and with wrapper.

TABLE III: Resource utilization of one PE

Xilinx zc7020		W/O wrapper		With NoC & wrapper	
Resources	Available	Used	%	Used	%
Slice registers	106400	568	1%	2795	2%
Slice LUTs	53200	1502	2%	3346	2%
DSP48E	220	1	1%	20	9%

VI. Case Study: Matrix Vector Multiplication over GF(2)

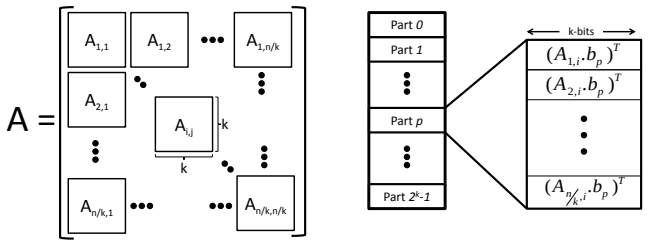
Integer factorization is one important application of Matrix Vector Multiplication over GF(2) and solutions have been proposed [10], [11] scaling over about a 1000 chips (ASICs, FPGAs). Block Wiedemann [12] algorithm is often used for this purpose, which needs computations of the form $(AV, A^2V, ..., A^rV)$ involving a very large boolean matrix A, and where V has more than one column vectors. Note that A is reused over all the iterations (r).

(The sparse floating-point version of the same problem would also have made a good case study, however, over GF(2), the approach used here is particularly communication intensive and through this we also show the impact of the choice of topology.)

A. Method: Sub-quadratic algorithm to BMVM

Our approach is based on the recently proposed combinatorial algorithm for matrix vector multiplication by Ryan Williams [5]. This approach involves a one-time pre-processing step on A, enabling a sub-quadratic time computation of BMVM.

The one-time pre-processing phase involves partitioning the matrix A into tiles of dimensions $k \times k$ as in Figure 13a, followed by construction of n/k look-up tables $\{\, LUT_i \mid i : 1 \rightarrow n/k \,\}$ corresponding to each of the n/k columns of the tiled A. LUT_i stores all possible linear combinations of columns of each $k \times k$ tile in the column i of the tiled matrix A (Figure 13a). There can be 2^k linear combinations of columns of each $k \times k$ tile, and there are n/k such tiles in a column of A. Figure 13b shows the composition of LUT_i, which is



(a) Tiling matrix A (b) Composition of the LUT_i

Fig. 13: One-time pre-processing phase

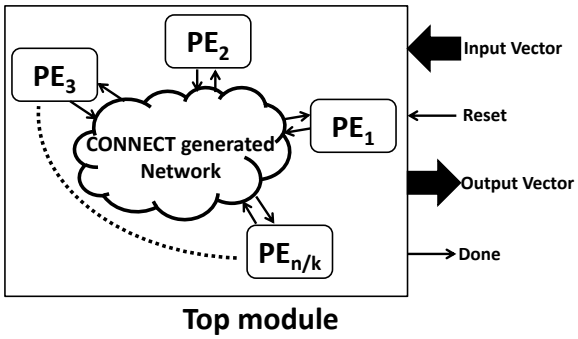


Fig. 14: Top module for Boolean Matrix Vector Multiplication (BMVM).

partitioned into 2^k parts, each part storing n/k $k-$bit words such that part$-p$ stores vectors $\{A_{1,i}b_p, A_{2,i}b_p, ..., A_{n/k,i}b_p\}$, where b_p is the k-bit vector corresponding to the partition index p. In short, the pre-processing step is equivalent to pre-computing and storing all possible products of the tiles of matrix A (ie., $A_{1,1}, A_{1,2}..A_{n/k,n/k}$) with any k-bit vector.

The computing phase uses this pre-processed information to compute Av, for some vector v. Let v be likewise partitioned into n/k *sub-vectors* $(v_1^T, v_2^T, .., v_{n/k}^T)$, and let $v' = Av = (v_1'^T, v_2'^T, .., v_{n/k}'^T)$. For illustration, let LUT_i, and v_i^T be with processing node-i (or thread-i). As $v_i' = A_{i,1}v_1 \oplus A_{i,2}v_2 \oplus \ldots \oplus A_{i,n/k}v_n/k$, if each processing node-i looks-up partition indexed by v_i in LUT_i, and send each of the n/k words stored in this partition to the corresponding processing nodes, the result v_i' at each processing node-i is obtained by XOR-accumulating all the incoming k-bit messages.

B. Implementation Details

The one-time precomputed LUTs are mapped to BRAMs on FPGA (Virtex 6 has about $38Mb$). Depending on the problem parameters (n and k), not all processing nodes can be mapped to a single FPGA. As per our earlier discussion, we map all the n/k processing elements across all the FPGAs in our NoC-driven multi-FPGA platform. It is important to ensure that while multiple such messages may simultaneously attempt to update a particular product sub-vector v_i', the updates are appropriately serialized to maintain correctness. Since only one flit can be injected and ejected in a single cycle in the NoC, this constraint is automatically ensured. Our implementation uses the following "Network and Router Options" for NoC generated using CONNECT (topology and number of endpoints specified as required):

Router Type	Simple Input Queued (IQ)
Flow Control Type	Peek Flow Control
Flit Data Width	16
Flit Buffer Depth	8
Allocator	Separable Input first Round-Robin

Since number of sub-vectors can be very large (n/k), we also implement "folding" (a folding factor f), such that a single processing element handles multiple sub-vectors and is provided with a single coalesced look-up table corresponding to the input sub-vectors. We use RIFFA 2.0 [13] to make this acceleration available to the software on the host.

C. Experimental Results

TABLE IV: Comparative results for $n = 64$ (64×64 Matrix) and $k = 8, (fold)f = 2$ (average over 100 experiments). Uses 4 PEs for the hardware and 4 threads for the software version.

Iterations r	Time (in msec)		Speedup
	Software	Mesh	(over s/w)
1	0.32	0.052	6.15
10	1.1	0.052	21.15
100	5.2	0.087	59.8
1000	44.2	0.58	76.2

TABLE V: Comparative results for $n = 1024$ (1024×1024 Matrix) and $k = 4, (fold)f = 4$ (average over 100 experiments). Uses 64 PEs (and 64 threads for s/w version).

Iterations	Time (in msec)				
	Software	Ring	Mesh	Torus	Fat_tree
1	4.0	0.205	0.075	0.060	0.052
10	22.9	1.67	0.412	0.299	0.275
100	204.3	16.15	3.64	2.83	2.33
1000	2025.4	160.51	35.60	28.09	22.69

The evaluation was done on Xilinx Virtex 6 ML605 on an Intel i7 host, hardware-software link between them was implemented using RIFFA 2.0 [13]. The multithreaded message passing software version (processing elements corresponding to threads) was evaluated on a 6 core Xeon (E5-2620). We compare the speed-up from the hardware-software solution compared to this multithreaded pure-software version of the algorithm. The hardware part on the FPGA operates on a 100 MHz clock.

Tables IV and V compare the performance of the multi-threaded message passing software model vs. its equivalent NoC realization on hardware (the times reported for this include the roundtrip time over RIFFA.) In Table V we have evaluated the results for four network topologies implemented on a single FPGA with single cycle hop between adjacent routers, which depict a clear correlation between network cost and performance (the cost increases moving from ring to mesh to torus to fat tree but performance also improves accordingly). When number of iterations are low (1-10), the

overheads in terms of host processor - FPGA communication time in hardware and thread *creation/join* time in software, are a dominant component of the overall execution time. For larger iterations (100-1000), the actual computation times dominate and the total execution time increases nearly linearly with number of multiplication iterations.

VII. Conclusion

We presented a semi-automated framework, complementary to existing HLS infrastructure, for scaling algorithms across multiple FPGAs. Through this work-in-progress, we share our experiences evaluating this process with three case studies, each of a different flavor. The application is expressed in the message passing abstraction, and realized over a Network-on-Chip. The network-on-chip abstraction is then extended automatically to seamless work across multiple FPGAs. The proof-of-concept evaluation was done between Xilinx Zynq FPGA (zed)boards.

VIII. Acknowledgements

This work was partially supported by Dr. Suhas Pai (through IITB Heritage Fund) and Nvidia (through CCOE, IITB).

References

[1] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *41st Annual International Symposium on Computer Architecture (ISCA)*, June 2014.

[2] Xilinx Inc., "Vivado HLS," 2013.

[3] W. J. Dally, C. Malachowsky, and S. W. Keckler, "21st century digital design tools," in *Proceedings of the 50th Annual Design Automation Conference*, p. 94, ACM, 2013.

[4] J. Porwal, S. Diwale, V. Kumar, and S. Patkar, "Projective geometry and precedence constraint based application mapping on multicore network-on-chip systems," in *VLSI Design, Automation and Test (VLSI-DAT), 2014 International Symposium on*, pp. 1–4, April 2014.

[5] R. Williams, "Matrix-vector multiplication in sub-quadratic time:(some preprocessing required)," in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 995–1001, Society for Industrial and Applied Mathematics, 2007.

[6] M. K. Papamichael and J. C. Hoe, "CONNECT: Re-examining conventional wisdom for designing NoCs in the context of FPGAs," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pp. 37–46, ACM, 2012.

[7] Y. Kou, S. Lin, and M. P. C. Fossorier, "Low-density parity-check codes based on finite geometries: a rediscovery and new results," *IEEE Transactions on Information Theory*, vol. 47, pp. 2711–36, Nov. 2001.

[8] H. Sharma, S. Das, R. R. Raut, and S. Patkar, "High throughput memory-efficient VLSI designs for structured LDPC decoding," in *PECCS*, pp. 518–521, 2011.

[9] P. Engineer, R. Velmurugan, and S. Patkar, "Parameterizable FPGA framework for particle filter based object tracking in video," in *28th International Conference on VLSI Design, VLSID 2015, Bangalore, India, January 3-7, 2015*, pp. 35–40, 2015.

[10] W. Geiselmann and R. Steinwandt, "Hardware to solve sparse systems of linear equations over gf (2)," in *Cryptographic Hardware and Embedded Systems-CHES 2003*, pp. 51–61, Springer, 2003.

[11] S. Bajracharya, D. Misra, K. Gaj, and T. El-Ghazawi, "Reconfigurable hardware implementation of mesh routing in number field sieve factorization," in *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on*, pp. 263–270, IEEE, 2004.

[12] D. Coppersmith, "Solving homogeneous linear equations over (2) via block wiedemann algorithm," *Mathematics of Computation*, vol. 62, no. 205, pp. 333–350, 1994.

[13] M. Jacobsen and R. Kastner, "Riffa 2.0: A reusable integration framework for fpga accelerators," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pp. 1–8, IEEE, 2013.

DSL-based Design Space Exploration for Temporal and Spatial Parallelism of Custom Stream Computing

Kentaro Sano
Graduate School of Information Sciences,Tohoku University
6-6-01 Aramaki Aza Aoba, Sendai 980-8579, JAPAN
Email: kentah@caero.mech.tohoku.ac.jp

***Abstract*—Stream computation is one of the approaches suitable for FPGA-based custom computing due to its high throughput capability brought by pipelining with regular memory access. To increase performance of iterative stream computation, we can exploit both temporal and spatial parallelism by deepening and duplicating pipelines, respectively. However, the performance is constrained by several factors including available hardware resources on FPGA, an external memory bandwidth, and utilization of pipeline stages, and therefore we need to find the best mix of the different parallelism to achieve the highest performance per power. In this paper, we present a domain-specific language (DSL) based design space exploration for temporally and/or spatially parallel stream computation with FPGA. We define a DSL where we can easily design a hierarchical structure of parallel stream computation with abstract description of computation. For iterative stream computation of fluid dynamics simulation, we design hardware structures with a different mix of the temporal and spatial parallelism. By measuring the performance and the power consumption, we find the best among them.**

I. Introduction

Recently, FPGA-based custom computing has been attracting a lot of application developers especially in the big-data and supercomputing fields, where not only performance but also power consumption is a very important. Since it is difficult to further increase a clock frequency of a general-purpose microprocessor, so far many-core accelerators such as GPUs have been considered as a promising solution to obtain higher computing performance. However, achievable performance is limited by the overall power budget of an entire system, and therefore power efficiency is considered as a key to large-scale computation.

On the other hand, custom computing with FPGAs is expected to provide comparable performance at much lower power consumption. Custom circuits are able to effectively achieve high performance by exploiting spatial and temporal parallelism of computing problems at a low clock frequency. Moreover, recent advancement of FPGAs fabricated by cutting-edge semiconductor technologies is bringing high potential for efficient and high performance computation due to on-chip integration of many hard macros such as block RAMs, high-speed I/O blocks, and DSP blocks. Especially emerging state-of-the-art FPGA devices are capable of very high performance numerical computation at a low power with their hard floating-point DSP blocks [7].

Stream computing is one of the promising approaches for efficient computation with custom hardware. This is because 1) deep pipelines can increase the number of operations performed per memory access, and 2) regular accesses for streaming data fully utilize a precious bandwidth of external memories. In addition, dedicated hardware designs bring efficient utilization of resources on FPGAs by adaptively giving a various mix of different operators and functions, including an adder, a multiplier, a divider, and a square root function. So far, researches have been reported on their successful high-performance stream computing with FPGAs [6], [9].

However, productivity still remains as a big issue not only in designing custom hardware, but also in exploring design space to obtain the best performance per power. In the case of stream computation, we can exploit the two types of parallelism: spatial, and temporal. By duplicating a pipeline to exploit the spatial parallelism, we can increase operations performed at every cycle, resulting in higher performance until all the available hardware resources are consumed. However, this also increases bandwidth requirements to an external memory, and the scalability is limited by the available bandwidth. On the other hand, by deepening a pipeline to exploit the temporal parallelism, we can increase operations per memory access, resulting in higher performance with the same memory bandwidth. However, too long pipelines suffer from low utilization due to the prologue and epilogue effects in pipelining. Thus, the best mix of the two different parallelism depends on these constraints, and therefore we have to find the optimal one for individual application.

In this paper, we present a domain-specific language (DSL) based approach to easily explore a design space for spatially and/or temporally parallel stream computation with FPGA. Our own DSL, called a stream processing description (SPD), allows us to intuitively describe formulae and submodule calls for various computation and structures of custom hardware in a software-like abstraction level. In design exploration, we design various parallel-configurations in SPD to be compiled with our SPD compiler, and then find the best among them by evaluating performance and power consumption of their actually working implementations with FPGA.

So far, several languages are proposed for stream processing, including StreamIt [11], and its parallelism is studied [3]. There are also presented stream computing compilers targeting FPGAs, non-commercial ones [4], [5] and commercial ones [1], [8]. Our DSL is designed for compact and intuitive description of hierarchal and modular connection of hardware modules for explicit parallelism. In this study, we apply it to design space exploration for high-performance custom computing of the scientific numerical simulation. Contributions of this work are:

1) DSL, called SPD for custom stream computation,
2) Framework for DSL-based design space exploration,
3) Case study for FPGA-based fluid dynamics simulation.

The SPD compiler used in this work is an extended version of the previous one published in [10]. The extension was made

2nd International Workshop on *FPGAs for Software Programmers (FSP 2015)*, London, United Kingdom, September 1, 2015.

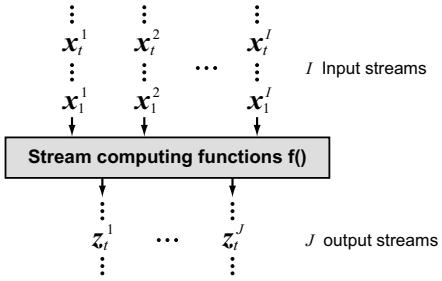
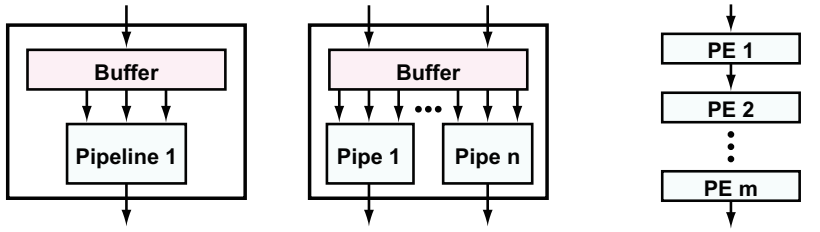


Fig. 1. Definition of stream computing.



a) PE with a single pipeline. b) PE with n pipelines. c) m cascaded PEs.

Fig. 2. Processing element (PE) (a), spatial parallelism (b), and temporal parallelism (c).

essentially for hierarchical and modular description capability.

This paper is organized as follows. Section II describes parallelism of iterative stream computation, and SPD for design of stream-computing hardware. Section III gives a design of an application example and evaluation. Finally, Section IV gives conclusions and future work.

II. DSL-based Design Space Exploration

A. Stream computation

Here we define stream computation which is targeted by our domain-specific language (DSL). Stream computation has I input data streams $\boldsymbol{x}^i$ $(1 \leq n \leq I)$ and J output data streams $\boldsymbol{z}^j$ $(1 \leq m \leq J)$, each of which has elements as follows:

$$\boldsymbol{x}^i \equiv \{\ \boldsymbol{x}_1^i,\ \boldsymbol{x}_2^i,\ ...,\ \boldsymbol{x}_t^i,\ ...,\ \boldsymbol{x}_T^i\ \}, \quad (1)$$

$$\boldsymbol{z}^j \equiv \{\ \boldsymbol{z}_1^j,\ \boldsymbol{z}_2^j,\ ...,\ \boldsymbol{z}_t^j,\ ...,\ \boldsymbol{z}_T^j\ \}. \quad (2)$$

Each input or output stream has T scalar elements, incoming or outcoming in order of a time $t = 1, 2, ..., T$. As shown in Fig.1, we model stream computation with a function $f^j()$ for the j-th output stream:

$$\boldsymbol{z}_t^j = f^j\left(\{..., \boldsymbol{x}_t^1, ...\}, \{..., \boldsymbol{x}_t^2, ...\}, ..., \{..., \boldsymbol{x}_t^I, ...\}\right), \quad (3)$$

which means that the output of the j-th stream at a time t is obtained by computing a given function with the input elements at t and their offset ones if necessary. For example, stencil computation of a single variable with a 3×3 star stencil on a 256×256 grid can be streamed with

$$\boldsymbol{z}_t = f\left(\boldsymbol{x}_{t-256},\ \boldsymbol{x}_{t-1},\ \boldsymbol{x}_t,\ \boldsymbol{x}_{t+1},\ \boldsymbol{x}_{t+256}\right). \quad (4)$$

B. Spatial and temporal parallelism

In this research, we focus on iterative stream computation, which is usually seen in time-marching simulation to repeatedly perform the same stream computation for time integral. For iterative stream computation, we can exploit both spatial and temporal parallelism by using processing elements (PEs) with multiple pipelines. Here we assume that a PE updates the entire data for a single time step by streaming them. Fig.2a is a processing element of stream computation, where we use the internal buffer for offset references of streamed data. In this case where only a single PE with a single pipeline is used, no coarse grain parallelism is exploited while fine grain parallelism is available with operators in the pipeline.

By duplicating the pipeline inside the PE as shown in Fig.2b, we can exploit spatial parallelism, or data parallelism to speed up computation for a single time step. Here we share the buffer with the n pipelines to restrict the increase of the buffer size. We fuse independent buffers into a single buffer with multiple inputs and outputs, so that most of the internal memories can be shared. When there is no dependency among computations of data stream elements, we can utilize this spatial parallelism to increase the performance with a similar size of a buffer. However, this approach requires more memory bandwidth due to the n times wider data stream.

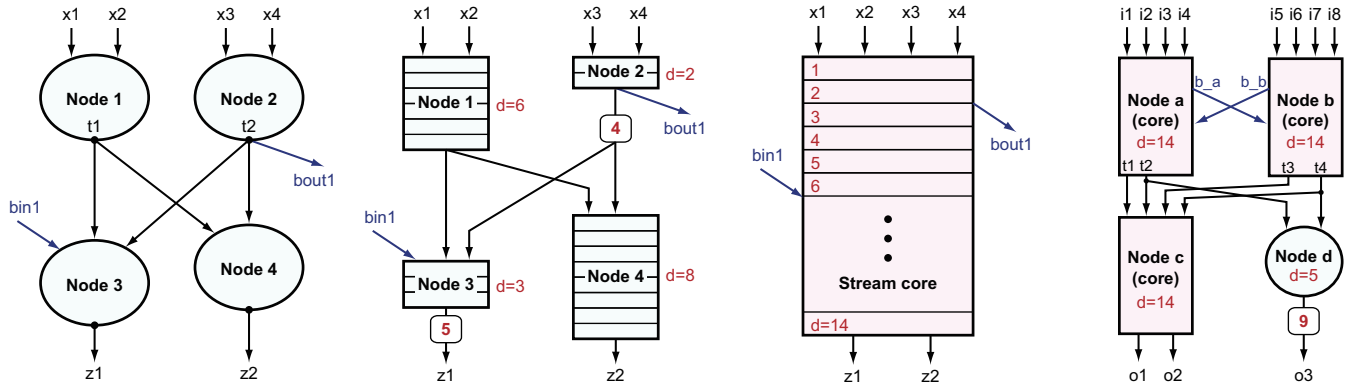
To the contrary, we can keep the same bandwidth requirement in increasing the performance by temporal parallelism. As shown in Fig.2c, we can cascade m PEs and use them as a longer pipeline to speed up computation for m time steps. The cascaded PEs require no wider bandwidth to stream data with an external memory because memory accesses are made only at the top and the bottom of the pipeline. Since streaming T data elements through d pipeline stages takes $(T + d)$ cycles, m-cascaded PEs take $(T + md)$ cycles while a single PE takes $m(T + d)$ cycles for computing m time steps. When $T >> d$, m times faster computation is achieved by cascading m PEs.

However, this approach has two inherent drawbacks. First, the total buffer size increases. Cascade connection of m independent PEs consumes m times more memories for their internal buffers. Accordingly, on-chip memory resources can limit the number of PEs cascaded when each buffer is large. Second, the utilization of PEs becomes lower due to the prologue and epilogue effects of a pipeline. In pipelining, the performance gain comes from parallel processing with different pipeline stages. Accordingly, some PEs are idle until all the PEs receive data elements to compute at the beginning of pipelining, and after PEs finish computation of the last element. The total effective performance can be much degraded when a short stream goes through a long pipeline.

We can apply both the temporal and spatial parallelism by cascading m PEs with n internal pipelines, giving a design space for various combinations of (n, m). On-chip hardware resources constrain available combinations while their performance depends on several factors including an external memory bandwidth, the depth of pipelines, and the size of stream data. Therefore, given a computing application and an FPGA board, we have to find the one for the best performance and power consumption among available combinations of (n, m).

C. Stream processing description (SPD)

It is not an easy task to explore design space by designing and implementing various hardware structures in RTL. To improve productivity of design space exploration, we propose a domain-specific language (DSL) for abstracted description of stream-computing hardware. We name the DSL "stream processing description", or SPD. We design SPD for the two major requirements. The first one is to easily and intuitively describe computations with formula, like software codes. The second one is to describe hardware structures in a simple way.



a) DFG of stream computing. b) Node pipelines connected with delays. c) Stream computing core constructed as a complete pipeline. d) Higher-level structure with cores for more complex computation.

Fig. 3. Hierarchical pipeline construction for stream computing with a data-flow graph (DFG).

```
1: Name         core;                    # name of this core
2: Main_In      {main_i::x1,x2,x3,x4};   # main stream in
3: Main_Out     {main_o::z1,z2};         # main stream out
4: Brch_In      {brch_i::bin1};          # branch inputs
5: Brch_Out     {brch_o::bout1};         # branch outputs
6:
7: Param          c  = 123.456;          # define parameter
8: EQU   Node1,   t1 = x1 * x2;          # eq (4) (Node1)
9: EQU   Node2,   t2 = x3 + x4;          # eq (5) (Node2)
10: EQU  Node3,   z1 = t1 - t2 * bin1;   # eq (6) (Node3)
11: EQU  Node4,   z2 = t1 / t2 + c;      # eq (7) (Node4)
12: DRCT      (bout1) = (t2);            # port connection
```

Fig. 4. Stream-processing description (SPD) code for DFG in Fig.3a.

```
1: Name         Array;
2: Main_In      {main_i::i1,i2,i3,i4,i5,i6,i7,i8};
3: Main_Out     {main_o::o1,o2,o3};
4:
5: HDL Node_a, 14, (t1,t2)(b_a) = core(i1,i2,i3,i4)(b_b);
6: HDL Node_b, 14, (t3,t4)(b_b) = core(i5,i6,i7,i8)(b_a);
7: HDL Node_c, 14, (o1,o2)      = core(t1,t2,t3,t4);
8: EQU Node_d,          o3      = t2 * t4;
```

Fig. 5. Stream-processing description (SPD) code for the structure in Fig.3d.

The SPD format allows us to intuitively describe computing formula for pipelines of PEs and connection of the PEs.

In the rest of this section, we use the following example of stream computation with an input vector $\boldsymbol{v}_i^{\text{in}} = (x1, x2, x3, x4)$ and an output vector $\boldsymbol{v}_i^{\text{out}} = (z1, z2)$:

$$t1 = x1 \times x2, \quad (5)$$
$$t2 = x3 + x4, \quad (6)$$
$$z1 = t1 - t2 \times b_{in1}, \quad (7)$$
$$z2 = t1 \ / \ t2 + c, \quad (8)$$
$$b_{out1} = t2, \quad (9)$$

where $t1$ and $t2$ are temporary variables, and c is a constant. b_{in1} and b_{out1} are additional input and output streams, respectively. A stream-computing hardware can be implemented as a static mapping of a data-flow graph (DFG) of computation.

1) Computation description: Fig.3a shows the DFG of computation for Eqs.(5) to (8), which correspond to Nodes 1 to 4, respectively. Thus, each node represents each formula. The directed edges show the dependences among the formulae. Eq.(9) is an output of Node 2 as `bout1`. The computation of a formula can be implemented as a pipelined data-path. Fig.3b shows pipelines for the DFG where nodes are replaced with their pipelined data-paths. Since nodes of different formulae can have a different number of pipeline stages, we have to equalize all the path lengths by inserting additional delays.

Figs.4 is an example description in SPD for the DFG of Fig.3a. We describe the computations only with 12 lines. Please note that strings after '#' are treated as comments. Each line is described in a common style of `"Function Fields"` for one of the functions summarized in Table I. These functions are mainly classified into "*core and interfaces*" and "*nodes and connection*." In the example code of Fig.4, Lines 1 to 5 are for the former. Line 1 names this core with `core`. Line 2 makes a main stream input interface with a name of `main_i` and its port names of `x1`, `x2`, `x3`, and `x4`. Similarly, Line 3 makes a main stream output interface `main_o` with ports `z1` and `z2`. Lines 4 and 5 make a branch input `brch_i` with a port `bin1`, and an output `brch_o` with a port `bout1`, respectively.

The remaining lines are written for nodes and connection. Line 7 beginning with `Param` defines a parameter `cnst` with a constant of 123.456, which is used in the formula of Line 11. Such parameters in formulae are statically replaced with their values by a preprocessor. Lines 8 to 11 create Nodes 1 to 4 for a static single assignment to an output port variable with a calculation formula. We refer to this type of a node as *an equation node* or simply *EQU node*. Function `EQU` is followed by an unique node name and a form of a port variable, '=', and a formula. In a formula, we can use parentheses, operators of `+`, `-`, `*`, and `/`, and a square root function of `sqrt()`.

In SPD, variables are 32-bit words. For EQU nodes, all related variables are treated as single precision floating-point numbers for numerical computation. When an output variable of a node is referred in a formula of another node, these nodes are connected with a directed edge in the DFG. In addition, we can explicitly connect ports of variables with different names by using `DRCT` function. Line 12 connects output `t2` of Node2 to input `bout1` of the branch output interface of Line 5.

2) Hardware structure description: The DFG mapped to hardware with pipelined nodes can be considered a single pipeline, as shown in Fig.3c, which can be used as a node in a DFG. This means that we can construct a higher level structure by connecting existing module cores. For example, in Fig.3d, the core of Fig.3c is used as the three nodes connected with

TABLE I. FORMAT OF STREAM PROCESSING DESCRIPTION (SPD).

Category	Function	Fields	Description
Core and interfaces	`Name`	`<core name>`	Set a name of this core.
	`Main_In`	`{<IF name>::port1, port2, ...}`	Append input ports for a main stream interface.
	`Main_Out`	`{<IF name>::port1, port2, ...}`	Append output ports for a main stream interface.
	`Brch_In`	`{<IF name>::port1, port2, ...}`	Append input ports for a branch interface.
	`Brch_Out`	`{<IF name>::port1, port2, ...}`	Append output ports for a branch interface.
Nodes and connection	`EQU`	`<node name>, "equation"`	Append an equation node.
	`HDL`	`<node name>, <delay>, "module call", <param list>`	Append an HDL node.
	`DRCT`	`(destination port list) = (source port list)`	Connect ports of nodes directly.
Others	`Param`	`<parameter name> = <constant value>`	Define a parameter with a constant value.

TABLE II. FORMAT OF SPD SUBFIELD.

Subfield	Format
`"equation"`	`<output port> = "calculation formula using input port names"` `Example: out = ( in1 + in2 * ( t1 - t2 ) ) / in3 + sqrt( in4 )`
`"module call"`	`(main output ports)(branch output ports) = <module name>(main input ports)(branch input ports)` `Example: (o1,o2,o3)(do1,bo2) = MyModule(x1,x2,x3,x4,x5)(bi1,bi2,bi3)`

another node. Thus we can hierarchically build a hardware structure from a low-level design of data-paths in a core to a high-level design of core connection. This approach provides high productivity in implementing various configurations for parallel stream computation with PEs.

Function `HDL` is used to create a node with an existing module. We refer to this type of a node as *HDL node*. The pipeline delay of the HDL node has to be statically known in advance of compilation. In HDL line, `HDL` is followed by a node name, a pipeline delay, module-call description, and a parameter list which is directly passed to parameters of a Verilog-HDL module. The parameter list can be omitted if not necessary. As shown in Table II, "module call" has a similar style to a subroutine call in software, except that multiple output variables can be specified. Fig.5 shows an example of multiple output variables which are written in parentheses. For HDL nodes, all variables are basically treated as raw binary data of 32-bit words while an actual data type in processing depends on each HDL node.

D. Library modules for HDL node

We can make HDL nodes not only by calling modules described in other SPD codes, but also by using existing modules written in HDL. EQU nodes allow developers to easily describe numerical computation, while HDL nodes extend the function of SPD to arbitrary operations and controls beyond computation. We provide elementary HDL modules in a library that can be used in stream processing without writing Verilog-HDL. The library of the present version contains Synchronous multiplexer, Comparator, Eliminator, Delay, Stream forward, Stream backward, and 2D stencil buffer modules.

III. DESIGN AND EVALUATION

A. Overview

As a benchmarking application for DSL-based design space exploration, we chose 2D fluid dynamics simulation based on the lattice Boltzmann method (LBM) [2]. We describe stream computation of LBM in SPD for hierarchical hardware structures; sub-modules for computing stages, a PE consisting of the sub-modules, and cascade connection of the PE. We compile the SPD codes with our stream-computing compiler, which is an extended version of [10], to obtain HDL codes of a custom computing core.

We use an IP-based system integration tool, ALTERA Qsys in order to build a system-on-chip (SoC) common platform consisting of PCI-Express I/F, memory controllers, scatter-gather DMAs, and their interconnects on FPGA. We can easily embed the core generated by the SPD compiler into the system, while this process is not completely automated yet. We also developed a Linux driver and a library software for data transfer between a host program and the FPGA board, and control of stream computation on FPGA.

We compiled the system with the embedded core by using ALTERA Quartus II 14.1 compiler to generate a bitstream for ALTERA Stratix V 5SGXEA7N2 FPGA. We verify FPGA-based fluid dynamics computation with a TERASIC DE5-NET board by comparing the computational results with those by software-based computation. All the designed LBM cores operate at 180 MHz, while 512-bit width DDR3 memory controllers operate at 200 MHz. We evaluate area, active power of the FPGA board, pipeline utilization, and sustained performance per power for design space exploration.

B. 2D fluid dynamics simulation based on LBM

In the 2D fluid dynamics simulation based on LBM, we compute propagation and collision of fictive particles over a discrete lattice mesh for viscous fluid flow. The details of computation are available in [6]. The computing algorithm of LBM has the three stages of the collision calculation, the translation, and the boundary computation. We wrote SPD codes separately for sub-modules of these stages. Please note that we made three different SPD codes of the translation stage for x1, x2, and x4 parallel pipelines.

Next, we wrote SPD codes to make PEs with $n = 1$, 2, and 4 pipelines. Figs.6 and 8 show the SPD codes for PEs with x1 and x2 pipelines, respectively. Figs.7 and 9 are their compiled DFGs. Here the rounded rectangles are HDL nodes, including the collision calculation node: `uLBM_calc`, the x1 or x2 translation node: `uLBM_Trans2D`, and the boundary computation node: `uLBM_bndry`. The synthesized PEs have 855 and 495 pipeline stages, respectively. Finally, we wrote SPD codes to cascade m PEs. Figs.10 and 11 show the SPD codes for $m = 1$ and 2 cascaded PEs with $n = 1$ pipeline. Fig.12 shows the DFGs of $m = 1$ and 2 PEs with $n = 1$ pipeline. Finally, we implemented six designs for (n, m) = $(1, 1)$, $(1, 2)$, $(1, 4)$, $(2, 1)$, $(2, 2)$, and $(4, 1)$.

```
Name     PEx1;
Main_In  {Mi::if0_0,if1_0,if2_0,if3_0,if4_0,if5_0,if6_0,if7_0,if8_0, iat_0,
             sop,eop, one_tau,rho_in,rho_out};
Main_Out {Mo::of0_0,of1_0,of2_0,of3_0,of4_0,of5_0,of6_0,of7_0,of8_0, oat_0,
             sop,eop};

################ Calculation stage (x1 parallel)
HDL uCalc0, 90,
    (f0_0_c,f1_0_c,f2_0_c,f3_0_c,f4_0_c,f5_0_c,f6_0_c,f7_0_c,f8_0_c) =
    Calc(if0_0,if1_0,if2_0,if3_0,if4_0,if5_0,if6_0,if7_0,if8_0, one_tau);

################ Translation stage (x1 parallel buffer)
HDL uTransx1, 724,
    (f0_0_t,f1_0_t,f2_0_t,f3_0_t,f4_0_t,f5_0_t,f6_0_t,f7_0_t,f8_0_t, at_0_t,
     Mo::sop,Mo::eop) =
     Transx2(f0_0_c,f1_0_c,f2_0_c,f3_0_c,f4_0_c,f5_0_c,f6_0_c,f7_0_c,f8_0_c,iat_0,
             Mi::sop,Mi::eop);

################ Boundary stage (x1 parallel)
HDL uBoundary0, 40,
    (f0_0_b,f1_0_b,f2_0_b,f3_0_b,f4_0_b,f5_0_b,f6_0_b,f7_0_b,f8_0_b,at_0_b) =
    Boundary(f0_0_t,f1_0_t,f2_0_t,f3_0_t,f4_0_t,f5_0_t,f6_0_t,f7_0_t,f8_0_t,at_0_t,
             rho_in,rho_out);

DRCT (of0_0,of1_0,of2_0,of3_0,of4_0,of5_0,of6_0,of7_0,of8_0) =
     (f0_0_b,f1_0_b,f2_0_b,f3_0_b,f4_0_b,f5_0_b,f6_0_b,f7_0_b,f8_0_b);
DRCT (oat_0) = (at_0_b);
```

Fig. 6. SPD code of a stream computing LBM PE with x1 pipeline.

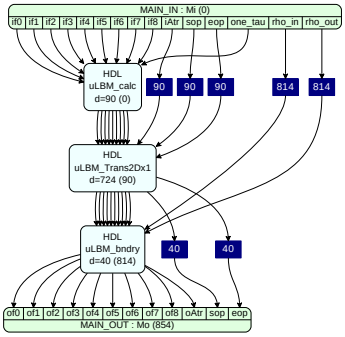


Fig. 7. Stream-computing LBM PE with x1 pipeline.

C. Resources and performance

Table III summarizes resource consumption of the implemented designs. The SoC peripherals including the PCI-Express I/F and DDR3 memory controllers consume about 23% of ALMs (adaptive logic modules), 6% of on-chip memories, and no DSP block. With the remaining resources, we implemented up to $nm = 4$ pipelines in PEs. Please note that for $nm = 4$, the four cascaded PEs with x1 pipelines consume 3.5 times more on-chip memories than those for the PE with x4 pipelines. This is because the x4 pipelines share a buffer which is slightly larger than the buffer for the x1 pipeline. However, in the case of this 2-dimensional LBM computation, the size of the buffer is very small and negligible. Each pipeline has a total of 131 floating-point (FP) operators in a single precision as shown in Table IV.

Let N_{Flops} denote the number of FP operators in each pipeline. Since m cascaded PEs with n pipelines perform nmN_{Flops} operations every cycle once a pipeline is filled, the peak performance is calculated with

$$P(n,m) = nmN_{\text{Flops}}F_{\text{GHz}} \quad \text{[GFlop/s]}, \tag{10}$$

where F_{GHz} is the operating frequency in GHz. In our designs, $F_{\text{GHz}} = 0.18$ and $N_{\text{Flops}} = 131$. Accordingly, the theoretical peak performance is 94.32 GFlop/s for $nm = 4$.

Then we evaluate the utilization of the PE pipelines. By using hardware counters inserted into the top of the LBM computing core, we counted the number of cycles (n_c) bringing valid data for computation, and the number of stall cycles (n_s) with no computation performed. We calculate the utilization u with $u = n_c/(n_c + n_s)$. As shown in Table III, the utilization is almost 1.0 for PEs with x1 pipeline while PEs with x2 or x4 pipelines have a much less utilization. This is because the DDR3 memory on the FPGA board has only 12.8 GB/s for each of read and write, which can support only the bandwidth required by the x1 pipeline, which is 7.20 GB/s.

```
Name     PEx2;
Main_In  {Mi::if0_0,if1_0,if2_0,if3_0,if4_0,if5_0,if6_0,if7_0,if8_0, iat_0,
             if0_1,if1_1,if2_1,if3_1,if4_1,if5_1,if6_1,if7_1,if8_1, iat_1,
             sop,eop, one_tau,rho_in,rho_out};
Main_Out {Mo::of0_0,of1_0,of2_0,of3_0,of4_0,of5_0,of6_0,of7_0,of8_0, oat_0,
             of0_1,of1_1,of2_1,of3_1,of4_1,of5_1,of6_1,of7_1,of8_1, oat_1,
             sop,eop};

################ Calculation stage (x2 parallel)
HDL uCalc0, 90,
    (f0_0_c,f1_0_c,f2_0_c,f3_0_c,f4_0_c,f5_0_c,f6_0_c,f7_0_c,f8_0_c) =
    Calc(if0_0,if1_0,if2_0,if3_0,if4_0,if5_0,if6_0,if7_0,if8_0, one_tau);
HDL uCalc1, 90,
    (f0_1_c,f1_1_c,f2_1_c,f3_1_c,f4_1_c,f5_1_c,f6_1_c,f7_1_c,f8_1_c) =
    Calc(if0_1,if1_1,if2_1,if3_1,if4_1,if5_1,if6_1,if7_1,if8_1, one_tau);


################ Translation stage (x2 parallel buffer)
HDL uTransx2, 364,
    (f0_0_t,f1_0_t,f2_0_t,f3_0_t,f4_0_t,f5_0_t,f6_0_t,f7_0_t,f8_0_t, at_0_t,
     f0_1_t,f1_1_t,f2_1_t,f3_1_t,f4_1_t,f5_1_t,f6_1_t,f7_1_t,f8_1_t, at_1_t,
     Mo::sop,Mo::eop) =
     Transx2(f0_0_c,f1_0_c,f2_0_c,f3_0_c,f4_0_c,f5_0_c,f6_0_c,f7_0_c,f8_0_c,iat_0,
             f0_1_c,f1_1_c,f2_1_c,f3_1_c,f4_1_c,f5_1_c,f6_1_c,f7_1_c,f8_1_c,iat_1,
             Mi::sop,Mi::eop);

################ Boundary stage (x2 parallel)
HDL uBoundary0, 40,
    (f0_0_b,f1_0_b,f2_0_b,f3_0_b,f4_0_b,f5_0_b,f6_0_b,f7_0_b,f8_0_b,at_0_b) =
    Boundary(f0_0_t,f1_0_t,f2_0_t,f3_0_t,f4_0_t,f5_0_t,f6_0_t,f7_0_t,f8_0_t,at_0_t,
             rho_in,rho_out);
HDL uBoundary1, 40,
    (f0_1_b,f1_1_b,f2_1_b,f3_1_b,f4_1_b,f5_1_b,f6_1_b,f7_1_b,f8_1_b,at_1_b) =
    Boundary(f0_1_t,f1_1_t,f2_1_t,f3_1_t,f4_1_t,f5_1_t,f6_1_t,f7_1_t,f8_1_t,at_1_t,
             rho_in,rho_out)();

DRCT (of0_0,of1_0,of2_0,of3_0,of4_0,of5_0,of6_0,of7_0,of8_0) =
     (f0_0_b,f1_0_b,f2_0_b,f3_0_b,f4_0_b,f5_0_b,f6_0_b,f7_0_b,f8_0_b);
DRCT (of0_1,of1_1,of2_1,of3_1,of4_1,of5_1,of6_1,of7_1,of8_1) =
     (f0_1_b,f1_1_b,f2_1_b,f3_1_b,f4_1_b,f5_1_b,f6_1_b,f7_1_b,f8_1_b);
DRCT (oat_0, oat_1) = (at_0_b, at_1_b);
```

Fig. 8. SPD code of a stream computing LBM PE with x2 pipelines.

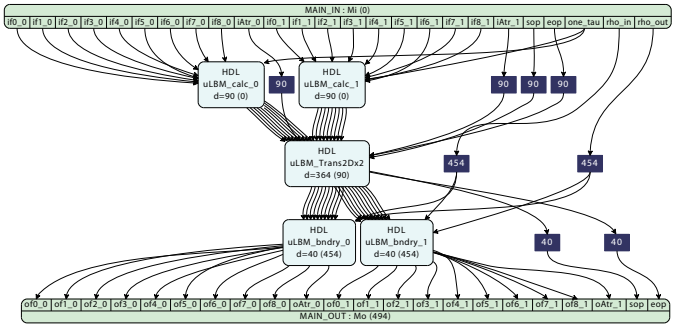


Fig. 9. Stream-computing LBM PE with x2 pipelines.

By multiplying the utilization and the peak performance, we obtain the sustained performance, which is shown in Table III. In our design space exploration, the configuration of $(n,m) = (1,4)$ gives the best sustained performance of 94.2 GFlop/s, which is very close to the peak. Please note that the negative effect on utilization in pipelining is negligible for a sufficiently large computing-grid, for example, a grid with 720×300 cells. To evaluate performance per power, we measured the power consumption of the FPGA board by measuring the power supplied by the PCI-Express edge connector with HIOKI power meter PW3336. The highest performance per power is also given by the same configuration of $(n,m) = (1,4)$, which is 2.4 GFlop/sW.

IV. Conclusions

This paper presents DSL-based design space exploration to find the best mix of the spatial and temporal parallelism

TABLE III. RESOURCE CONSUMPTION, OPERATING FREQUENCY, PIPELINE UTILIZATION, PERFORMANCE, AND POWER.

Device / Modules	ALMs	%	Regs	%	BRAM [bits]	%	DSPs	%	Freq.	Utilization	Performance	Power	Perf/W
Stratix V 5SGXEA7	234720	100	938880	100	52428800	100	256	100	[MHz]	(u)	[GFlop/s]	[W]	[GFlop/sW]
SoC peripherals	54997	23.4	87163	9.28	3110753	5.93	0	0.0	-	-	-	-	-
(n pipelines, m PEs) = (1, 1)	34310	14.6	62145	6.62	573370	1.09	48	18.8	180	0.999	23.5	28.1	0.837
(1, 2)	63687	27.1	122426	13.0	1243564	2.37	96	37.5		0.999	47.1	30.6	1.542
(1, 4)	129738	55.3	244196	26.0	2987730	5.70	192	75.0		0.999	94.2	39.0	2.416
(2, 1)	64119	27.3	122630	13.1	642410	1.23	96	37.5		0.557	26.3	32.3	0.812
(2, 2)	136742	58.3	244195	26.0	1316604	2.51	192	75.0		0.558	52.6	37.4	1.405
(4, 1)	128431	54.7	243626	25.9	859604	1.64	192	75.0		0.279	26.3	33.2	0.792

```
Name       mQsys_Core10;
Main_In  {Mi::if0_0,if1_0,if2_0,if3_0,if4_0,if5_0,if6_0,if7_0,if8_0,iAtr_0,sop,eop};
Main_Out {Mo::of0_0,of1_0,of2_0,of3_0,of4_0,of5_0,of6_0,of7_0,of8_0,oAtr_0,sop,eop};
Append_Reg {Mi::one_tau, rho_in, rho_out};  ## Definition of constant inputs

################ PEx1_1
HDL Core_1, 495,
    (f0_0_1,f1_0_1,f2_0_1,f3_0_1,f4_0_1,f5_0_1,f6_0_1,f7_0_1,f8_0_1,Atr_0_1,
     sop_1,eop_1) =
     PEx1(if0_0,if1_0,if2_0,if3_0,if4_0,if5_0,if6_0,if7_0,if8_0,iAtr_0,
          Mi::sop,Mi::eop, one_tau,rho_in,rho_out);

DRCT (of0_0, of1_0, of2_0, of3_0, of4_0, of5_0, of6_0, of7_0, of8_0) =
     (f0_0_1,f1_0_1,f2_0_1,f3_0_1,f4_0_1,f5_0_1,f6_0_1,f7_0_1,f8_0_1);
DRCT (oAtr_0, Mo::sop, Mo::eop) = (Atr_0_1, sop_1, eop_1);
```

Fig. 10. SPD code of a single PE with x1 pipeline.

TABLE IV. THE NUMBER OF FLOATING-POINT OPERATORS IN A CORE.

	Adder	Multiplier	Divider	Total
PE with x1 pipeline	70	60	1	**131**

for FPGA-based iterative stream computation. To allow to intuitively describe formulae and submodule calls for various computation and structures of custom hardware in a software-like abstraction level, we propose a domain-specific language, called SPD. Although the SPD compiler is not completely automated yet for the exploration, it allows software developers to design and implement custom hardware with various parallel configurations more easily than doing in conventional RTL languages. Evaluating six configurations of stream-computing cores for fluid dynamics simulation based on LBM, we found the best performance per power is obtained by the design depending only on the temporal parallelism due to the memory bandwidth requirement less than the available one on the used FPGA board.

In the future work, we will automate the process of design space exploration with software codes of a target application.

ACKNOWLEDGMENTS

This research was supported by Grant-in-Aid for Challenging Exploratory Research No.23650021 from the Ministry of Education, Culture, Sports, Science and Technology, Japan.

REFERENCES

[1] "Altera Corp. WEB," http://www.altera.com/.

[2] N. David R, S. Chen, J. G. Georgiadis, and R. O. Buckius, "A consistent hydrodynamic boundary condition for the lattice boltzmann method," *Physics of Fluids*, vol. 7, no. 1, pp. 203–209, January 1995.

[3] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 151–162, October 2006.

[4] A. Hagiescu, W.-F. Wong, D. F. Bacon, and R. Rabbah, "A computing origami: Folding streams in FPGAs," *Proceedings of the Design Automation Conference*, pp. 282–287, July 2009.

```
Name       mQsys_Core10;
Main_In  {Mi::if0_0,if1_0,if2_0,if3_0,if4_0,if5_0,if6_0,if7_0,if8_0,iAtr_0,sop,eop};
Main_Out {Mo::of0_0,of1_0,of2_0,of3_0,of4_0,of5_0,of6_0,of7_0,of8_0,oAtr_0,sop,eop};
Append_Reg {Mi::one_tau, rho_in, rho_out};  ## Definition of constant inputs

################ PEx1_1
HDL Core_1, 495,
    (f0_0_1,f1_0_1,f2_0_1,f3_0_1,f4_0_1,f5_0_1,f6_0_1,f7_0_1,f8_0_1,Atr_0_1,
     sop_1,eop_1) =
     PEx1(if0_0,if1_0,if2_0,if3_0,if4_0,if5_0,if6_0,if7_0,if8_0,iAtr_0,
          Mi::sop,Mi::eop, one_tau,rho_in,rho_out);

################ PEx1_2
HDL Core_2, 495,
    (f0_0_2,f1_0_2,f2_0_2,f3_0_2,f4_0_2,f5_0_2,f6_0_2,f7_0_2,f8_0_2,Atr_0_2,
     sop_2,eop_2) =
     PEx1(f0_0_1,f1_0_1,f2_0_1,f3_0_1,f4_0_1,f5_0_1,f6_0_1,f7_0_1,f8_0_1,Atr_0_1,
          sop_1,eop_1, one_tau,rho_in,rho_out);

DRCT (of0_0, of1_0, of2_0, of3_0, of4_0, of5_0, of6_0, of7_0, of8_0)  =
     (f0_0_2,f1_0_2,f2_0_2,f3_0_2,f4_0_2,f5_0_2,f6_0_2,f7_0_2,f8_0_2);
DRCT (oAtr_0, Mo::sop, Mo::eop) = (Atr_0_2, sop_2, eop_2);
```

Fig. 11. SPD code of two cascaded PEs with x1 pipeline.

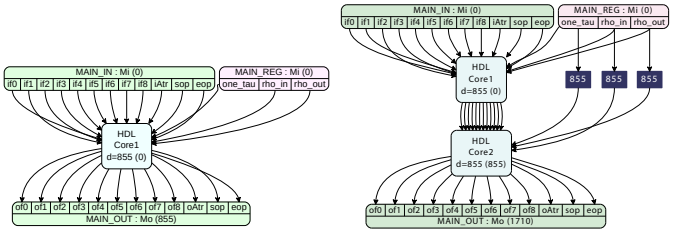


Fig. 12. A single PE and two cascaded PEs with x1 pipeline.

[5] A. Hormati, M. Kudlur, S. Mahlke, D. Bacon, and R. Rabbah, "Optimus: Efficient realization of streaming applications on FPGAs," *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pp. 41–50, October 2008.

[6] Y. Kono, K. Sano, and S. Yamamoto, "Scalability analysis of tightly-coupled FPGA-cluster for lattice boltzmann computation," *Proceedings of the 22nd International Conference on Field-Programmable Logic and Applications*, August 2012, paper W3B1.

[7] M. Langhammer and B. Pasca, "Floating-point DSP block architecture for FPGAs," *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 117–125, February 2015.

[8] O. Pell, O. Mencer, K. H. Tsoi, and W. Luk, *High-Performance Computing Using FPGAs*. Springer New York, July/August 2013, ch. Maximum Performance Computing with Dataflow Engines, pp. 747–774.

[9] K. Sano, Y. Hatsuda, and S. Yamamoto, "Multi-FPGA accelerator for scalable stencil computation with constant memory-bandwidth," *IEEE Transaction on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 695–705, March 2014.

[10] K. Sano, H. Suzuki, R. Ito, T. Ueno, and S. Yamamoto, "Stream processor generator for HPC to embedded applications on FPGA-based system platform," *Proceedings of the International Workshop on FPGAs for Software Programmers*, pp. 43–48, September 2014.

[11] W. Thies, M. Karczmarek, and S. Amarasinghe, "Streamit: A language for streaming applications," *Proceedings of the 11th International Conference on Compiler Construction*, pp. 179–196, April 2002.

Coarse-Grain Performance Estimator for Heterogeneous Parallel Computing Architectures like Zynq All-Programmable SoC

Daniel Jiménez-González*†, Carlos Álvarez*†, Antonio Filgueras†, Xavier Martorell*†,
Jan Langer‡, Juanjo Noguera‡ and Kees Vissers‡
*Universitat Politecnica de Catalunya
†Barcelona Supercomputing Center
Email: {daniel.jimenez,carlos.alvarez,antonio.filgueras,xavier.martorell}@bsc.es
‡Xilinx USA
Email: {jan.langer,juanjo.noguera,kees.vissers}@xilinx.com

***Abstract*—Heterogeneous computing is emerging as a mandatory requirement for power-efficient system design. With this aim, modern heterogeneous platforms like Zynq All-Programmable SoC, that integrates ARM-based SMP and programmable logic, have been designed. However, those platforms introduce large design cycles consisting on hardware/software partitioning, decisions on granularity and number of hardware accelerators, hardware/software integration, bitstream generation, etc.**

This paper presents a performance parallel heterogeneous estimation for systems where hardware/software co-design and run-time heterogeneous task scheduling are key. The results show that the programmer can quickly decide, based only on her/his OmpSs (OpenMP + extensions) application, which is the co-design that achieves nearly optimal heterogeneous parallel performance, based on the methodology presented and considering only synthesis estimation results. The methodology presented reduces the programmer co-design decision from hours to minutes and shows high potential on hardware/software heterogeneous parallel performance estimation on the Zynq All-Programmable SoC.

I. Introduction

With the end of Dennard Scaling [7] computer architecture has entered a new era. One main thread followed by several architectures has been to evolve into multi- and many-core systems composed of several identical cores. Another important trend has been the incorporation of large, specialized accelerator systems (mainly evolved from the graphics ecosystem) that efficiently execute single instruction mulitple thread codes.

However, the struggle to squeeze some performance out of a continuously growing number of transistors per chip has, somehow, avoided its most obvious and promising path: the creation of a large number of very specialized accelerators that can, on one side, be really energy efficient and, on the other, make the work faster by eliminating software overheads. Indeed, ASICs, so common only a few years ago, are being progressively discarded in favor of cheaper, more general components that have the essential advantage of short time-to-market cycles. In this sense, new hybrid CPU-FPGA systems can be seen as the future of heterogeneous computing. While being roughly one order of magnitude slower that its ASIC equivalent, FPGAs can be reprogrammed on the fly, and adapt to changing environments. Furthermore, being tightly integrated with general cores, those systems can retain the programmability of common CPUs and join it with the tremendous boost in performance and efficiency that characterizes specialized hardware.

However, in order to be broadly used in the mass-market, those systems still face two important challenges: first, a software ecosystem that facilitates their programmability without burdening the programmer with all the cumbersome details (data transfers, synchronization, memory coherence...) of heterogeneous systems and, second, an easy and fast way to perform a quick decision of the best mapping of all the application components to the most adequate hardware to compute them in a parallel heterogeneous system.

Several works have addressed the first problem and it can be said that parallel programming models (like OmpSs [9] and OpenMP 4.0) can be used to solve it. However the second problem is still a barren field. Indeed, assuming that the first step of selecting which application kernels can be computed by the reconfigurable hardware is done by a programmer, and generating the proper HDL code for them could be done automatically, still remains the problem that a FPGA bitstream generation can take several hours. After that, the whole application should be analyzed only to find out if the hardware-software partition, or the resources distribution between the kernels in the FPGA, are adequate or not for the scheduling policy of the programming model and the heterogeneous system at hand. Any mistake in the selection or any bad guess by the expert programmer means repeating the whole process leading to a trial and error process composed of several hours steps..., in addition to a unexpected parallel heterogeneous performance.

In this paper we address those issues by presenting a way to speed-up the process and help introduce heterogeneous systems into everyday computation. The suggested workflow is designed to estimate the performance of OmpSs applications on any heterogeneous parallel architecture using the execution time information (estimated or not) of the OmpSs tasks running on the processing components of the target architecture. Those heterogeneous architectures are currently target by cyber-physical computing platforms [2], [5] that may combine cluster of nodes with SMP, FPGAs and GPUs. Here, we focus on the

2nd International Workshop on *FPGAs for Software Programmers (FSP 2015)*, London, United Kingdom, September 1, 2015.

heterogeneous parallel performance estimation for the Zynq All-Programmable SoC architecture, that combines ARM-based SMP and a FPGA, and that also includes GPUs in the next generation Zynq UltraScale+ MPSoC[20]. This workflow integrates a simulator which implements the runtime of the OmpSs programming model, and a shared memory coarse-grain component (ARM cores and FPGA accelerators) architecture with local memory (BRAM) for the FPGA accelerators. This simulation is fed by the reports obtained from the high level synthesis tool for the timing information of the hardware accelerators considered by the programmer and a task-based trace generated by a sequential execution of the OmpSs application. The framework simulates the execution of the trace tasks in a data-flow manner as the software runtime of OmpSs would do, considering all the components of the heterogeneous architecture. With this, the simulator can obtain the estimated heterogeneous parallel performance for the application kernels (tasks) in the target system. Finally, once the best alternative is selected based on the simulator results, the bitstream can be effectively generated and executed in order to check the correctness of the conclusions. This process lasts just a few minutes or even seconds to achieve similar results than having to generate the bitstream for each possible mapping and run the application in the real system.

Our methodology currently considers that the programmer is an parallel programmer that only needs to explore few hardware/software codesigns, otherwise a design space exploration strategy should be analyzed to reduce the amount of possible solutions, like using back annotations [11], [19].

So, the contributions of the paper are as follows:

- Light coarse-grain estimation that helps the programmer to have a fast order-of-magintude decision of the best hardware/software co-design on a heterogeneous parallel system with FPGA devices, reducing the number of bitstreams to be generated.
- *Heterogeneous parallel performance* estimator based on a task-based trace driven simulator that integrates the runtime of the OmpSs programming model and a shared memory coarse-grain component architecture, with local memory for the hardware accelerators.
- A complete framework that avoids the need of placing and routing each FPGA accelerator required by the programmer annotated tasks with target FPGA.

The rest of this paper is organized as follows: Section II presents the related work of the paper. After that section III presents the methodology suggested, section IV its implementation and section V presents the experimental setup. Section VI presents the results obtained. Finally, section VII concludes the article.

II. Related Work

The work presented in this paper addresses the problem of how to use efficiently, at run-time, the hardware resources of an heterogeneous system selecting the best among the large amount of implementation possibilities that such a system offers. To do so, it relies in the existence of a programming model that allows the integrated programming of heterogeneous systems, predicting the performance of the applications on those systems. Meanwhile there are several programming models dealing with FPGA-based heterogeneous systems [15], [6], [1], [10], [16], [8], there has been less work in the literature regarding the performance prediction of heterogeneous applications. Indeed, most of those works cover the prediction of a kernel performance on an FPGA [13], [14], [18] and only few of them analyze full application performance prediction. The RC amenability test (RAT) [12] is a system that tries to predict the suitability of an application to be ported to an FPGA, in order to avoid the work if the outcome is foreseeable as non successful. Another performance prediction technique [17] addresses the modelling of shared heterogeneous workstations containing reconfigurable computing devices. This methodology chiefly concerns the modelling of system level, multi-FPGA architectures. However, it does not take into account the selection among several different application kernels or the interactions between them in a given parallel application.

Other works propose electronic system level timing and power estimation that combines system-level timing and power estimation techniques with platform-based rapid prototyping [11], [19]. However, the annotated task has to be specified in a particular language and/or has to be mapped to a specific component of the system. In our work, the same task can be annotated in C/C++ language with OpenMP-like task directives, which is the standard shared memory programming model. In addition, tasks can be annotated to be mapped, at run-time, to different components of the heterogeneous system depending on the scheduling policy.

To the best of our knowledge the work presented in this paper is the only one that deals with those kernel selection and performance prediction challenges that require a *run-time* analysis of the application and the prediction of complex, irregular task dependency execution patterns.

III. Methodology

In this section we present the methodology proposed in this paper to reduce the developer effort to map complex applications to heterogeneous parallel systems with FPGAs. OmpSs programming model makes easy the programmability of applications with kernels (pieces of code, functions or not) that may be executed in both CPUs and FPGAs [10] as offloaded tasks, transparently to the programmer, by writing simple pragma directives. Note that, as commented above, the automatic generation of the different granularities and the architecture configurations (number of accelerators for each kernel) is beyond of the scope of this paper contribution. Therefore, the granularity of the tasks and target devices where those tasks could be run should be indicated by the programmer. The decision of where those tasks are executed is automatically done at run-time.

Figure 1 shows an example of an OmpSs blocking matrix multiplication. As it can be observed in the first line of the code, the `mxmBlock` kernel has been annotated as it can be executed in both FPGA and SMP. The second line of the code specifies that this function will become a OmpSs task (with input and output dependences) any time it is called in the code. With these annotations, the OmpSs runtime can take care of scheduling different instances of the kernel, when their dependences are ready, in both resources based on availability.

```
#pragma omp target device(fpga,smp)
#pragma omp task in([BS*BS]A,[BS*BS]B)\
                 inout([BS*BS]C)
void mxmBlock( REAL *A, REAL *B, REAL *C)
{
  int i, j, k;
  for (i=0; i < BS; i++)
    for (k=0; k < BS; k++) {
      REAL tmp = A[i*BS+k];
      for (j=0; j < BS; j++)
        C[i*BS+j] += tmp * B[k*BS+j];
    }
}

void matmul(REAL **AA,REAL **BB,REAL **CC,int NB)
{
  int i, j, k;
  for (k = 0; k < NB; k++)
    for(i = 0; i < NB; i++)
      for (j = 0; j < NB; j++)
        mxmBlock(AA[i*NB+k],BB[k*NB+j],\
                 CC[i*NB+j]);
}
```

Fig. 1. Matrix multiplication annotated with OmpSs directives. `matmul` is the blocking matrix multiplication function, and `mxmBlock` performs the matrix multiplication of a block.

However, even if the translation from C to HDL is done automatically with Vivado and OmpSs is used to schedule the `mxmBlock` tasks to the best available computing unit, the problem of how to partition the work remains. How many instances of the `mxmBlock` should be implemented in the available hardware? How big should be any of the instances? Is it worth to implement two instances of a different size? Indeed, in the presence of several possible kernels to be mapped to the FPGA when all of them will not fit, which ones have to be mapped to maximize the application performance? The expert programmer may have an idea of which is the best combination and reduce the number of possible implementations to few of them (tens). However, those few implementations may mean hundreds of hours if each of them implies one or more bitstream generations.

In order to answer those questions a coarse-grain performance estimator toolchain has been developed. This performance estimator toolchain combines instrumentation based on source to source compilation, high level synthesis from C code and a heterogeneous task-based dataflow parallel simulator developed to estimate the heterogeneous parallel performance of OmpSs code with heterogeneous tasks. This simulator has Extrae [3] instrumentation support that allows it to generate Paraver [4] traces, allowing the programmers to have an approximate visualization of what one would expect in a real task execution on an heterogeneous system. Extrae is a instrumentation library that generates time events, thread states and communications in a raw trace that can be translated to a Paraver trace format. In this work we have integrated the simulation with a modified version of the Extrae so that Extrae can take the timing of the simulation. Paraver is a tool that allows performance analysis of parallel execution traces at different levels of granularity: thread, task, MPI process, etc.

Figure 2 shows the overall steps of the developed

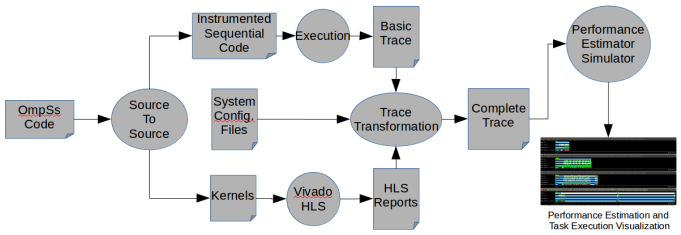


Fig. 2. Coarse-Grain Performance Estimator Toolchain

toolchain. The parallel programmer has to provide the OmpSs code with the annotation of the tasks and the granularity he/she wants to evaluate. As we mentioned above, the parallel programmer should have an idea of which are the most potential combinations of tasks, reducing the amount of possible task mappings and granularities. The automatic generation of the different granularities is beyond of the scope of this paper contribution. However, a starting programmer may need to analyze a large number of granularities and mappings, and in this case, a sytem to automatize the design space exploration would be helpful [11].

The first step of the toolchain is to (1) transform the OmpSs code to a sequential instrumented code, and (2) extract the kernel code of each task annotated by the programmer. Both these steps are automatically performed by a source to source compiler from the original OmpSs code. Once the instrumented sequential code has been obtained it is executed in order to obtain a trace of tasks that will be used for the performance estimation. The information contained in the trace will be joined with hardware timing information (estimated cycles and clock frequency) obtained from passing the extracted kernel codes through Vivado HLS and fed to our heterogeneous performance estimator that simulates the dynamic behavior of a preconfigured system (a particular implementation of the application in the Zynq board in our case) and returns not only the estimated time used by the given application in the selected hardware configuration but also a Paraver trace that can be visualized in order to further analyze the possible bottlenecks of the design. The whole cycle only takes few minutes and can be repeated as many times as necessary until all the possibilities have been explored. Finally, the best implementation can be chosen and the time consuming process of the hardware bitstream generation is done only once. The next section further explains how the performance estimation is done accurately enough to obtain useful results for the programmer's hardware/software co-design decision.

IV. Implementation

In order to obtain enough information from the OmpSs code, this is first transformed into an instrumented sequential code by source to source compilation. During this transformation, the directives of OmpSs are replaced with instrumentation of the tasks to be able to generate a task execution trace that will contain the following basic information: task number, creation time and elapsed execution time in cycles in the CPU based machine, number of dependences of the task, and for each dependence: the data dependence memory address and a label indicating the direction (input, output or inout) of the dependence, and finally, task name for later identification in the performance estimator toolchain.

The basic trace, generated by the execution of the instrumented sequential code, should be completed with further information. First of all, the cost of creation of a task has to be added. Each OmpSs task has a creation cost that is not generated by the instrumented sequential code. Therefore, each task instance of the task execution trace needs to be preceded by its creation cost (creation cost task), that will be run (in the simulation) only in the SMP device (independently if the task is executed in the FPGA or in the SMP). The original task instance in the trace will depend on the new creation cost task. Next, the information of the devices where each task can be executed and the latency of those should be also added. With this objective, the extracted kernels are used in order to obtain the latency of the hardware accelerators of those tasks that can be run, based on programmer annotation, in the FPGA. The latencies estimated for the computation and the input and output transfers are obtained by passing the extracted task code through the Vivado HLS, which, in few seconds, can generate the HDL code and a report with all the information required for this task code:

- Estimated number of cycles of the computation of the task in the FPGA
- Estimated number of cycles spent transferring the input/output parameters of the task to the FPGA

Using that information, each of the task instances that appears in the basic trace is completed with more information that states that the task can be also run in a hardware accelerator, and with the latency of the associated hardware accelerator.

Further specific information, related to the system where the programmer wants to execute the OmpSs code, should be taken into account to complete the trace. For instance, it should be evaluated if the system can overlap input and/or output DMA memory transfers (between the shared memory and the local memory in the accelerators) among different hardware accelerators. However, this analysis only needs to be done once. If the transfers can be done in parallel the input and output data transfer latencies can be added to the computational latency of the hardware accelerator associated to this task. Otherwise, DMA memory transfer tasks will be created and run in a shared hardware resource device to avoid possible overlapping of input/output transfers. Those extra tasks will have dependences with the corresponding tasks run in the device. In the case of our target architecture, the Zynq 706 board, and the current environment analyzed, the input parameters seem to scale with the number of accelerators, but not the output parameters. Figure 3 shows the speedup obtained when using 2 accelerators compared to 1 accelerator to transfer the same amount of input and output memory data: 512Kbytes and 1024Kbytes. Therefore, the time associated with a task running in a hardware accelerator device can be seen as the time of the input data DMA transfer plus the computation time. This information, together with the time this task lasts in a SMP core, will be part of the information of this task in the trace. However, the output DMA memory transfer cost will be represented by an new transfer task that will be run in a shared hardware resource device to avoid possible overlapping of output transfers, since no overlapping seems to be allowed. This output transfer will have an dependence with the original tasks run in the device.

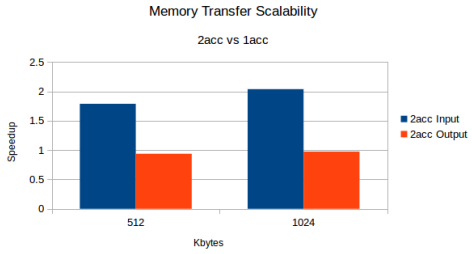


Fig. 3. Speedup of using 2 accelerators vs 1 accelerator for the input/output data transfers on the Zynq 706 Board for two different amounts of data.

On the other hand, each of those DMA transfers has to be programmed in software from the SMP device. This software cost may not be able to be done in parallel since they have to use shared resources. Then, DMA programming tasks (submit tasks) that will be run in a special device, shared among all the hardware accelerators, are created for each input/output transfer. The original task will depend on the input submit tasks and the output submit tasks will depend on the original task.

Once the trace has been completed with all the above information, the heterogeneous parallel architecture performance estimator can simulate the execution of all the tasks (original, creation cost, and DMA related tasks) in a dataflow manner for a given configuration of the hardware. That is, it will take care about the task input, output, and inout dependences and will run them as soon as their dependences are ready and a device that can execute them is available. The task dependency management is done the same way that the OmpSs runtime software system does. The performance estimator toolchain can be run, for an specific transformed trace, under different hardware configurations based on the programmer annotations. As a result, the Paraver traces generated by the estimator will allow the programmer to choose the best estimated combinations of software and hardware accelerators.

V. Experimental Setup

Results in Section VI have been obtained on a Zynq All-Programmable SoC 706 board. Timing of the applications has been obtained by instrumenting with `gettimeofday` the part of the code that calls several times the kernel code. Results show the average elapsed execution time of 10 application executions on the Zynq 706 board under linux.

The OmpSs implementation is based on Mercurium 1.99.4 and Nanos++ 0.8. For the hardware compilation branch we have used the Xilinx ISE Design 14.7 and the Vivado HLS 2013.2 tools. The estimator has been developed with support for Extrae Library 2.5.1 and Paraver 4.3.5. The Paraver was used to analyze the estimated execution traces. All OmpSs codes have been compiled with the `arm-xilinx-linux-gnueabi-g++ (Sourcery CodeBench Lite 2011.09-50) 4.6.1` and `arm-xilinx-linux-gnueabi-gcc (Sourcery CodeBench Lite 2011.09-50) 4.6.1` compilers, with `"-O3"` optimization flag.

We show real execution and estimator results for 2 tiled applications: matrix multiply (Figure 1) and cholesky (Figure 4), using different `fpga` task granularities for the tiles

(blocks): 64×64-block single-precision floating point matrix multiply (fine-grained tasks), 128×128-block single-precision floating point matrix multiply and 64×64-block double-precision floating point cholesky decomposition. In the case of the cholesky decomposition three out of four of the kernels are annotated to be able to be run in the SMP and also the FPGA. The fourth one has not been considered to be mapped to the FPGA by the programmer. All real data generated by the Vivado HLS has been synthesized with IEEE-754 standard compliance.

```
#pragma omp target device(fpga,smp)
#pragma omp task in([BS*BS]A) inout([BS*BS]C)
void dsyrk(double *A, double *C, int BS);

#pragma omp task inout([BS*BS]A)
void dpotrf( double *A, int t, int BS );

#pragma omp target device(fpga,smp)
#pragma omp task in([BS*BS]A) inout ([BS*BS]B)
void dtrsm( double *A, double *B, int t, int BS);

#pragma omp target device(fpga,smp)
#pragma omp task in([BS*BS]A, [BS*BS]B)\
                inout ([BS*BS]C)
void dtrsm( double *A, double *B, double *C,\
            int t, int BS);

void chol_ll(double **AA, int t, int NB, int BS)
{
  for (int k = 0; k < NB; k++ ) {
    for (int j=0; j<k; j++)
      dsyrk(AA[j*NB+k], AA[k*NB+k], BS);

    dpotrf(AA[k*NB+k], t, BS );

    for (int i = k+1; i < NB; i++)
      for (int j=0; j<k; j++)
        dgemm(AA[j*NB+i],AA[j*NB+k],\
              AA[k*NB+i],t,BS);

    for (int i = k+1; i < NB; i++)
      dtrsm(AA[k*NB+k], AA[k*NB+i], t, BS );
  }
}
```

Fig. 4. Cholesky application annotated with OmpSs directives. Each of the function calls will be a task instance (`dsyrk, dtrsm, dtrsm`: SMP and FPGA, `dpotrf`: SMP only).

VI. Results

In this section, a coarse-grain comparison of the estimator and real execution results is shown. The comparison is done varying relevant aspects in the design of heterogeneous parallel applications for FPGA based architectures, highlighting the analysis time required in our proposed methodology.

In the case of the tiled matrix multiplication we show a performance estimation that evaluates three different possible design decisions. The first one is to select between two different task granularities (64x64 blocks and 128x128 blocks) for the task kernel `mxmBlock` in Figure 1. Second, it is evaluated the difference between using one or two accelerators for running 64x64 `mxmBlock` tasks. Having two accelerators for the 128x128-block case has not been considered in the evaluation because the hardware resource estimation for two 128x128-block `mxmBlock` accelerators indicates that it is not feasible to map them into the programmable logic. Finally, we have considered the performance impact of allowing heterogeneous execution (`mxmBlock` task is specified with SMP and FPGA) or not.

Figure 5 shows the performance results of the commented cases for both the estimator and the real execution. Results are normalized with respect to the slowest case (one accelerator of 128x128 blocks and with heterogeneous execution - label `1acc 128 + smp` in the figure). Although estimator and real execution have different absolute speedups (our estimator does not consider memory hierarchy aspects like cache coherence and pinning of memory pages, neither memory contention, etc.), results show the same speedup trends. That allows the programmer to adapt her/his OmpSs program to have 128x128 `mxmBlock` tasks with the FPGA as the only target device. This decision can be taken after less than 5 minutes of work (coffee break), that is what the analysis requires under the proposed methodology. Figure 6 shows the time (seconds)

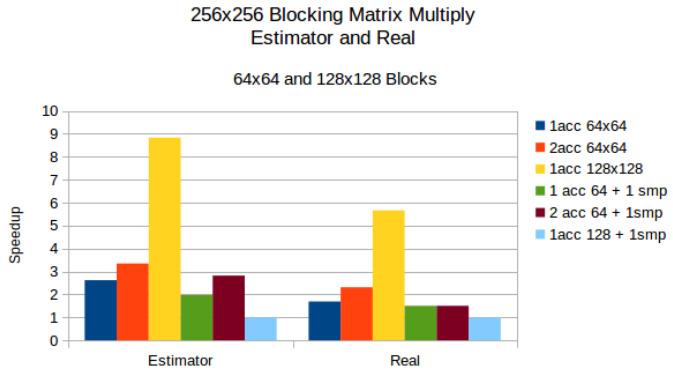


Fig. 5. Estimation and real matrix multiply performance comparison for different hardware configurations of the system and task configurations.

in logarithmic scale for the analysis of the configurations under our proposed methodology (left) and the traditional hardware-software design cycle (right). In particular, for the traditional design cycle, we only count the hardware generation of the different accelerators and combinations. The hardware generation time required for the full-analysis is more than 10 hours. On the other hand, the performance estimator toolchain lasts for less than 5 minutes and automatically provides the best choice among the considered configurations.

In addition to this decision, the programmer may want to do a depth analysis of the performance estimation using the Paraver traces generated in the estimation process. Paraver traces can be visualized and compared to detect potential bottlenecks in the parallel and heterogeneous execution of the tasks. Figure 7 shows the Paraver view of four estimated task execution traces for four different configurations shown in Figure 5 with the same time scale; from top to bottom: 1 acc 128x128, 2 acc 64x64, 2 acc 64x64 + SMP and 1 acc 128x128 + SMP. Paraver traces show the execution of the tasks (original and additional ones) in the devices, along the time (x-axis). Each Paraver trace shows an horizontal bar for each of the devices. First horizontal bar shows SMP task executions

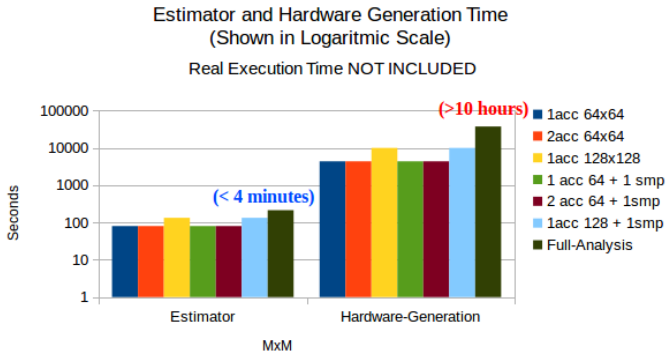


Fig. 6. Matrix Multiplication analysis time compared to hardware generation time of the hardware accelerators.

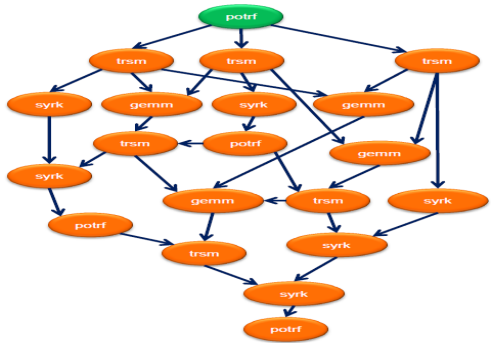


Fig. 8. Cholesky task dependecy graph for number of blocks equal to 4.

(original and creation tasks), last two bars show tasks running on shared locked resources (output DMA memory transfer from the FPGA, and DMA programming - submit) and the rest of the bars show `mxmBlock` tasks executed in the accelerators. The analysis shows that the current scheduling policy does not help to improve the performance when running `mxmBlock` in both SMP and FPGA. The high cost of executing the SMP version of the task compared to the FPGA version may be translated into a huge load imbalance problem if a wrong scheduler decision is taken. This has a significant impact in the case of 1 acc 128x128 running in both SMP and FPGA.

In the case of the tiled cholesky we have evaluated different resources distribution approaches between kernels that execute interleaved due to the complex nature of the cholesky dynamic graph (Figure 8). In order to simplify the example, the task granularity is fixed (64x64 blocks) and we have only evaluated which kernels should or should not be accelerated in the FPGA (note that to further complicate the scheduling, the application even has tasks - `dpotrf` task of Figure 4 - that can only be run in the SMP). As it is shown in Figure 9, the same speedup (normalized to the slowest configuration) trends are obtained in both estimated and real performance. The first three bars show the performance impact of implementing accelerators that try to maximize the usage of the hardware resources of the programmable logic (`FR-dgemm`, `FR-dsyrk`, `FR-dtrsm`, where `FR` stands for full resources), which limits the number of accelerators that fit in the hardware to one and forces all the other kernels to be executed in the SMP. The last set of three bars evaluate the performance of all the possible combinations of two tasks among three annotated with target FPGA (`dgemm+dgemm`, `dgemm+dsyrk`, `dgemm+dtrsm`) as the configuration only supports two accelerators.

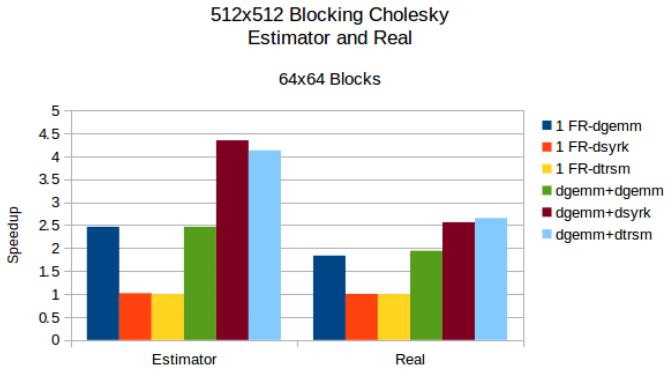


Fig. 9. Estimation and real cholesky performance comparison for different hardware configurations of the system and task configurations.

The programming productivity gain of the tiled cholesky is much more significant. A full analysis of those combinations requires one day and a half compared to less than 10 minutes with our methodology. Indeed, this day and a half is just for hardware generation time, no execution time is included neither creating the hardware design and integrating it to the rest of the application.

VII. Conclusions and Future Work

In this paper we have presented the current status of a heterogeneous parallel performance estimator that can help to potentially reduce the development effort in heterogeneous parallel computing systems like the Zynq All-Programmable SoC. The methodology is currently implemented for OmpSs applications and Zynq SoC. OmpSs is a task-based dataflow parallel programming model that helps to express heterogeneous task decompositions of an application. Thus, the programmer can annotate the application with OmpSs directives to identify tasks and the target devices where those tasks can be executed at run-time. Based on this information, our methodology estimates which is the best hardware-software partitioning of the annotated tasks on the Zynq Soc in few minutes. Results show that the best configurations and OmpSs annotations chosen by our estimation correspond with the real ones for the evaluated applications and configurations. And although the current performance estimator toolchain could be extended to automatically take care of different numbers of resources (e.g. the number of channels between the FPGA and the memory, cache coherence impact, etc), and explore different design space exploration strategies, the current implementation already shows speedups of more than two orders of magnitude (minutes vs days) on the process of achieving high heterogeneous performance for complex applications like cholesky.

Future work is to integrate power-efficiency and look-ahead scheduling heuristics into the simulator as well as helping the programmer with the hardware/software partitioning strategy to improve performance and/or area for a broader set of application domains.

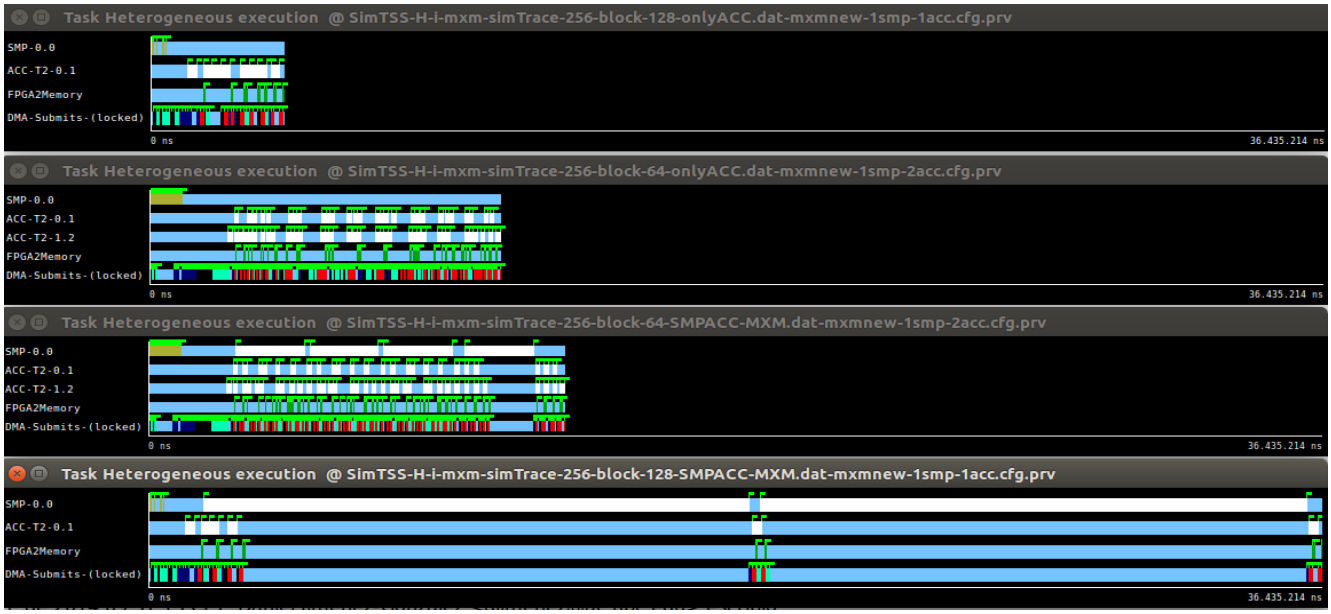


Fig. 7. MxM performance estimator traces for heterogeneous task executions running on 1 or 2 accelerators and none/one SMP. Blocksizes: 64x64 and 128x128.

VIII. Acknowledgments

We thank the anonymous referees for their valuable feedback. This work is supported by the AXIOM project, funded by EU H2020 program (grant ICT-01-2014 GA 645496), the Spanish Government, through the Severo Ochoa program (grant SEV-2011-00067) the Spanish Ministry of Science and Technology (TIN2012-34557) and the Generalitat de Catalunya (MPEXPAR, 2014-SGR-1051). We thank the Xilinx University Program for its hardware and software donations.

References

[1] Altera, Corp. *Nios II C2H Compiler User Guide*, 2009.

[2] C. Alvarez, E. Ayguad, J. Bueno, A. Filgueras, D. Jimnez-Gonzlez, X. Martorell, N. Navarro, D. Theodoropoulos, D. Pnevmatikatos N., D. Scordino, Catani, Claudio, P. Gai, C. Segura, C. Fernandez, D. Oro, J. Rodriguez-Saeta, P. Passera, A. Pomella, A. Rizzo, and R. Giorgi. The axiom software layers. In *DSD '15, Proceedings of the Euromicro Conference on Digital System Design, co-located with the 41st Euromicro Conference on Software Engineering and Advanced Applications (TO APPEAR)*, June 2015.

[3] Barcelona Supercomputing Center. *Extrae Instrumentation Library*, Sept. 2013. http://www.bsc.es/computer-sciences/extrae.

[4] Barcelona Supercomputing Center. *Paraver Visualization Tool*, Sept. 2013. http://www.bsc.es/computer-sciences/performance-tools/paraver.

[5] P. Burgio, C. Alvarez, E. Ayguad, A. Filgueras, D. Jimnez-Gonzlez, X. Martorell, N. Navarro, and R. Giorgi. Simulating next-generation cyber-physical computing platforms. In *De-CPS '15, Proceedings of the snd Workshop Challenges and New Approaches for Dependable and Cyber-Physical System Engineering (TO APPEAR)*, June 2015.

[6] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. Legup: High-level synthesis for fpga-based processor/accelerator systems. FPGA '11, pages 33–36, 2011.

[7] R. H. Dennard, F. H. Gaensslen, H. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9:256–268, Oct. 1974.

[8] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A hybrid multi-core parallel programming environment. In *First Workshop on General Purpose Processing on Graphics Processing Units*, October 2007.

[9] A. Duran, J. M. Perez, E. Ayguadé, R. M. Badia, and J. Labarta. Extending the openmp tasking model to allow dependent tasks. In *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism*, IWOMP'08, pages 111–122, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] A. Filgueras, E. Gil, D. Jimenez-Gonzalez, C. Alvarez, X. Martorell, J. Langer, J. Noguera, and K. Vissers. Ompss@zynq all-programmable soc ecosystem. FPGA '14, pages 137–146. ACM, 2014.

[11] K. Grüttner, P. A. Hartmann, T. Fandrey, K. Hylla, D. Lorenz, S. Stattelmann, B. Sander, O. Bringmann, W. Nebel, and W. Rosenstiel. An ESL timing & power estimation and simulation framework for heterogeneous socs. In *XIVth International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS 2014, Agios Konstantinos, Samos, Greece, July 14-17, 2014*, pages 181–190. IEEE, 2014.

[12] B. Holland, K. Nagarajan, C. Conger, A. Jacobs, and A. D. George. Rat: A methodology for predicting performance in application design migration to fpgas. In *Held in Conjunction with SC07*, HPRCTA '07, pages 1–10, 2007.

[13] T. Jeger, R. Enzler, D. Cottet, and G. Troster. The performance prediction model - a methodology for estimating the performance of an fpga implementation of an algorithm. In *Technical report*, 2000.

[14] D.-U. Lee, A. Gaffar, O. Mencer, and W. Luk. Optimizing hardware function evaluation. *Computers, IEEE Transactions on*, 54(12):1520–1531, Dec 2005.

[15] W. A. Najjar and J. R. Villarreal. Fpga code accelerators - the compiler perspective. In *DAC*, page 141, 2013.

[16] The Portland Group. *PGI Accelerator Programming Model for Fortran & C*.

[17] M. C. Smith and G. D. Peterson. Parallel application performance on shared high performance reconfigurable computing resources. *Performance Evaluation*, 60(1):107–125, 2005.

[18] C. Steffen. Parametrization of algorithms and fpga accelerators to predict performance. *Proc. Reconfigurable System Summer Institute (RSSI)*, pages 17–20, 2007.

[19] M. Streubhr, R. Rosales, R. Hasholzner, C. Haubelt, and J. Teich. Esl power and performance estimation for heterogeneous mpsocs using systemc. In *FDL*, pages 1–8. IEEE, 2011.

[20] Xilinx. Zynq UltraScale+ MPSoC, Aug. 2015. http://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html.

DESIGNING HARDWARE/SOFTWARE SYSTEMS FOR EMBEDDED HIGH-PERFORMANCE COMPUTING

Mário P. Véstias[†], *Rui Policarpo Duarte*[‡], *Horácio C. Neto*[⋆]

[†]INESC-ID, ISEL - Instituto Superior de Engenharia de Lisboa, Instituto Politécnico de Lisboa
[‡] Departamento de Ciência e Tecnologia, Universidade Autónoma de Lisboa
[⋆]INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal,
mvestias@deetc.isel.ipl.pt, rpduarte@ual.pt, hcn@inesc-id.pt

ABSTRACT

In this work, we propose an architecture and methodology to design hardware/software systems for high-performance embedded computing on FPGA. The hardware side is based on a many-core architecture whose design is generated automatically given a set of architectural parameters. Both the architecture and the methodology were evaluated running dense matrix multiplication and sparse matrix-vector multiplication on a ZYNQ-7020 FPGA platform. The results show that using a system-level design of the system avoids complex hardware design and still provides good performance results.

I. INTRODUCTION

Computing requirements of embedded systems are rapidly increasing with stringent real-time requirements, together with low power and low cost. Single processor solutions are unable to provide the required performance and at the same time keep the power consumption low. Hardware/software architectures where the most computational demanding parts of the application run in dedicated hardware have shown very good performance, area and power efficiencies.

While efficient, hardware/software architectures are in general difficult to obtain since designing dedicated hardware for a specific algorithm in FPGAs requires hardware expertise. From the perspective of the software programmer, an automatic flow to design and configure the hardware/software architecture is essential.

In this work, our approach is to use a configurable hardware coprocessor whose design is generated automatically after being parameterized by the programmer. The coprocessor consists of a many-core architecture that is automatically generated and integrated with the embedded processor. The many-core coprocessor is configurable in the number of cores, the system memory (number and size of local memories, cache and interfaces to external memory) and the topology of the interconnection network (Network-on-Chip, ring or simply point-to-point connections). The number and type of arithmetic operations of each core, number formats, including floating-point and integer can also be configured. Each core has local memory, an arithmetic unit and input/output interfaces. Keeping the core simple permits to explore more parallelism, reduces power consumption and makes configuration easier. The design and programming of the architecture was integrated in a proposed design flow that starts with the algorithm specification and outputs the hardware/software system to be implemented in a SoC FPGA.

Constraining the hardware design space to a hardware template may reduce the performance compared to a fully-optimized solution. However, it typically provides a good tradeoff between hardware performance, hardware portability and design time.

The paper is organized as follows. Section 2 describes the state-of-the-art in tools and architectures to design hardware/software processing architectures for FPGAs. Section 3 describes the proposed hardware/software architecture. Section 4 describes the proposed architecture design flow. Section 5 shows the results obtained and section 6 concludes the paper.

II. RELATED WORK

Many commercial and academic tools have been proposed to raise the design synthesis level of FPGAs and therefore reduce the design time and design efforts. High-level synthesis tools exist to generate hardware from C/C++ (C-to-Verilog [5], Catapult-C [3], Mitrion-C [4], ImpulseC [2], HandelC [1], Xilinx AutoESL, etc), SystemC (Bluespec [6], Xilinx AutoESL), Java (JHDL [7], MaxCompiler [8]), Python (MyHDL [9]), among others.

The most common approach for hardware compilation is to start with C/C++, with some language restrictions to avoid recursions and pointers. Compiler techniques were proposed to generate an optimized design for specific hardware platforms. For some large designs the generated hardware obtained with these tools were able to achieve better optimized hardware implementations compared to hand-made solutions.

In these tools, additional annotations are used in the code to control some implementation options. The compil-

2nd International Workshop on *FPGAs for Software Programmers (FSP 2015)*, London, United Kingdom, September 1, 2015.

ers extract as much as possible instruction-level parallelism that can be exposed using techniques like loop unrolling and pipelining. These compilers automatically generate the hardware but programmers must be aware of the hardware programming model which requires some knowledge on circuit design.

Another research direction for hardware design consists of using overlays that implement an intermediate reconfigurable architecture within the user logic of the FPGA. In [10] and [12] programmable overlays are used to increase the performance of DSP workloads on FPGA. In [11] the viability of a GPU-like overlay for FPGA was analyzed. However, whether GPU-like programming models and architectures are a good way to design many-cores on FPGA is yet to be checked. Also, if not carefully designed these overlays will run with low sustained performances compared to their peak performances.

Our proposal for the design of hardware/software high-performance embedded systems is to consider the hardware side as a many-core coprocessor. The coprocessor architecture is configurable at a system-level where the programmer only has to specify system-level parameters, like, the number of cores, the numerical precision of the arithmetic units, among others. Each core runs from simple to complex arithmetic operations, like vector multiplication, matrix multiplication. These operations are part of a library and new operations can be added through microcode programming.

A few many-core designs on FPGA have already been proposed. The MPLEM system [13] consists of Xilinx MicroBlaze soft-core processors connected with On-chip Peripheral Bus (OPB) buses. In [14] a system with 24 MicroBlaze cores interconnected with an Arteris NoC [15] was proposed. The system was implemented in a Virtex-4 FX-140 FPGA.

Tumeo et al. [16] proposed a real-time many-core system for automotive applications also based on Microblaze. Each core contains local data memory and all cores share an external RAM for shared data and instructions. Cores can communicate through a bus-based shared memory, or a message-passing subsystem built upon a crossbar module.

HeMPS-based systems [17] are homogeneous multiprocessor platforms using a network-on-chip (NoC) interconnection. Each processing element has a Plasma processor [19], an internal RAM block, a network interface to the NoC and a DMA engine. The platform is automatically generated and the number of processors can be customized. Design space exploration is based on simulation. Processors are modeled using cycle accurate instruction set simulators and local memories with C/SystemC models.

MARC (Many-core Approach to Reconfigurable Computing) [18] is a many-core template comprising one control processor and multiple processors for running tasks as SIMD (single instruction multiple data) units. Cores can be configured as RISC processors or synthesized as full-custom datapaths. Each core has local private memory and have access to an internal shared memory. Processors are interconnected with a network selected from a library with various topologies, including crossbar and torus.

SMYLEref [21] is a many-core architecture for embedded systems prototyped in FPGA. The architecture consists of multiple clusters arranged in a two-dimensional array connected with a NoC. Each cluster has a number of scalar processors connected with a local bus. Each core has dedicated instruction and data L1 caches. A second layer of cache exists in each cluster shared by all cores. The processor core is a Geyser [22].

Most of these many-core proposals rely on general-purpose embedded processors as the core unit. This increases flexibility but decreases performance and area efficiency. In approaches, like MARC, it is possible to customize the processor with a dedicated datapath that requires hardware design, but the results are still far from the peak capacity of the FPGA. Design space exploration is not specified in most approaches, but HeMPS, for example, uses ISS and system level simulation models to explore different platforms.

In our architecture, the core elements are based on simple processing units with reduced control, small local memories and arithmetic units. Each core unit can be individually configured in terms of local memory size and number and type of arithmetic operations. This permits to improve performance and area efficiency when compared to many-core architectures based on general-purpose embedded processors. We consider a customizable interconnection network that can be a bus, a crossbar, a NoC or a ring, and that can use point-to-point connections and/or a mix of these topologies. We rely on SystemC to do the design space exploration. We model the many-core platform and the algorithm using SystemC and do system level simulations to help in design space exploration.

III. HARDWARE/SOFTWARE MANY-CORE ARCHITECTURE

The proposed hardware/software architecture consists of an embedded processor and the many-core architecture (see figure 1).

The many-core has access to external memory through a DMA that is configured by the embedded processor. The DMA is responsible for sending/receiving data to/from memory and for forwarding this data to the network. In order to improve the bandwidth when requesting elements stored non-sequentially in memory, the DMA has a cache to buffer bursts of data and thus enable faster access. Each time non-sequential data is requested from memory, a burst of sequentially-stored elements is fetched (cacheline size). The first element of the burst is the data requested. This data is immediately forwarded to the processors. The other elements are stored in cache.

The cores are organized in clusters. Each cluster has a local lite processor (local PE) to program the cores and

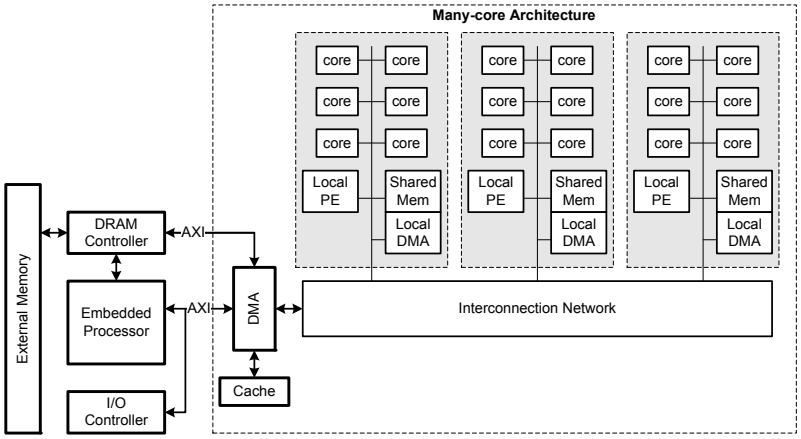


Fig. 1. Hardware/software many-core architecture

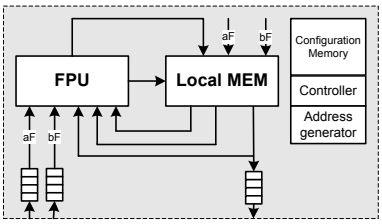


Fig. 2. Architecture of the core unit

the local DMA and control data communication; shared memory and a local DMA to transfer data to/from the cores. Cores are connected to the communication network through input and output buffers. Input buffers are connected to the FPU and to the local memory (*aF* and *bF* signals)

Each core has an arithmetic unit and a local data memory (see Figure 2). The arithmetic unit can be statically or dynamically configured to execute a set of basic functions: add/sub, multiplier, fused multiply-add, reciprocal, square root and inverse square-root [20]; and a set of more complex functions: vector multiplication, block matrix multiplication, etc. Each core can be configured with a different combination of operations and new complex operations can be added trough microprogramming. The local memory is implemented with dual-port block RAMs that are used to store temporary variables (registers are implemented with this memory), coefficients to implement some of the arithmetic operators, constants, and output data.

IV. ARCHITECTURE DESIGN FLOW

A many-core generator was developed to automatically generate the many-core architecture from a set of architecture specifications. Also, an instance of the many-core platform is also automatically modeled in SystemC for a system level simulation to determine the number of execution cycles considering different configurations of the architecture. In this version, the code to run on the cores

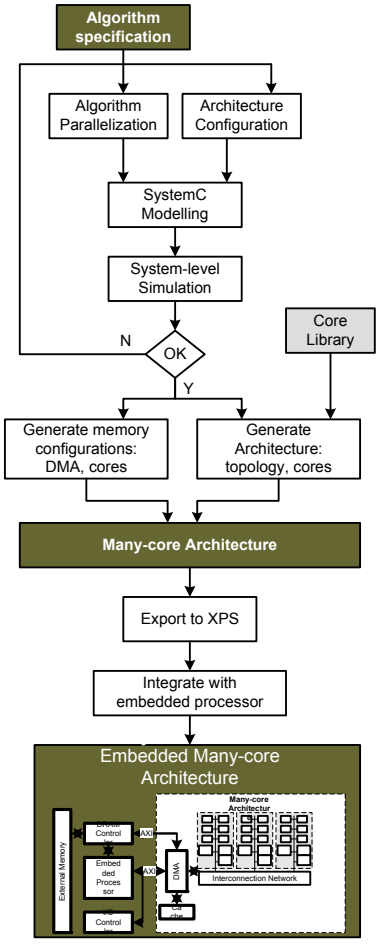


Fig. 3. Many-core design flow

and to program the DMA are obtained manually (see figure 3).

The flow starts with the configuration of the architec-

ture. The architecture may be simulated at system-level for a specific algorithm. To do this, a SystemC model of the architecture is automatically generated and then the algorithm must be parallelized, modelled in SystemC together with the architecture description. After these steps, the many-core architecture is generated using a library of cores and the software to run in the DMA and in the cores is manually generated by the designer. Cores are programmed with microcode instructions. The many-core is then exported to XPS (Xilinx Platform Studio) and integrated with the embedded processor present in the ZYNQ platform.

The design space exploration process is manual, that is, the designer is responsible for manually specifying different configurations of the many-core architecture and different algorithm parallelizations. The only automatic processes are the generation of a SystemC description of the architecture given a particular configuration of the architecture and algorithm, and the generation of a VHDL description of the many-core architecture to be synthesized and its integration in the XPS from Xilinx to generate the complete hardware/software embedded architecture. Along the flow the designer does not have to design hardware since the hardware is generated automatically.

V. RESULTS

To evaluate the flow and the architecture, we have considered parallel algorithms for dense matrix multiplication and sparse matrix-vector multiplication. For both, we explored the design space looking for the best many-core using the proposed flow. Both architectures were implemented in a ZYNQ-7000 SoC XCZ7020-CLG484 and tested on a ZedBoard with this device.

V-A. Configuration of the Architecture for Dense Matrix Multiplication

Matrix multiplication $C = A \times B$ is implemented as a parallel block matrix algorithm that partitions C matrix into smaller sub-matrices (blocks) and works with these blocks. All matrices are square and have the same size ($n \times n$).

The C matrix is divided in blocks with size $n \times xp$. Each of these blocks is calculated by p cores simultaneously. Each core is responsible for a sub block with size $n \times x$ which in turn is divided in smaller blocks with dimension $y \times x$. The size of these smaller blocks, C_{ij}, depends on the local memory size. To generate a block C_{ij} the processor multiplies a block $y \times n$ from matrix A with a block $n \times x$ from matrix B. The multiplication is implemented as a sequence of k partial block multiplications,

$$C_{ij} = \sum_{k=1}^{k_0} A_{ik} \times B_{kj} \quad (1)$$

Each partial block is the multiplication of a $y \times z$ sub block A_{ik} with a $z \times x$ sub block B_{kj}, resulting in a partial sub block C_{ij_k} of size $y \times x$. The final C_{ij} result is obtained after accumulating the k partial block results.

The partial block multiplications are implemented as follows. First, each core receives and stores its B_{qj} elements. Then, A_{iq} elements are broadcasted to all cores. As the A_{iq} elements arrive, they are multiplied by all B_{qj} elements stored in local memory. The partial results of each block C_{ij} are also stored in local memory. In the final iteration, the elements of the result block C_{ij} are sent to the external memory. As referred, the local memory in each processor must store the blocks of B (size $z \times x$) and C (size $x \times y$) under processing.

At the algorithmic level, x, y and z are variables and thus different performance results are obtained by changing these values. To optimize the final solution, we have considered the theoretical results in [25] to determine these values. According to the referenced theoretical results, the number of communications with the external memory does not depend on the dimension z of the sub blocks. Therefore, z can be simply made equal to 1 in order to reduce the local memory required. The local memory necessary to store the sub blocks of B (size $1 \times x$) is doubled in order to enable the processor to store a new B sub-block while still performing the computations with the former B sub-block.

Also according to this reference, the dimensions of the sub blocks C_{ij} that minimize the number of communications, as a function of the available local memory L, are

$$x = \frac{L}{2 + \sqrt{p\,L}} \quad y = \sqrt{p\,L} \quad (2)$$

At the architectural level, matrix multiplication requires multiply and add operations. So, the arithmetic units of all cores are configured as fused multiply-add. We have configured the many-core with 16 and 32 cores, all with the same local memory size and a DMA cache with support for up to 16 cachelines.

Assuming an architecture with 32 KBytes of local memory for the 16-core and 16 KBytes for the 32-core architecture, we have determined the utilization of resources and the number of execution cycles (see table I). Both architectures achieve high performance efficiencies (peak performance/measured performance), 86% and 84%, respectively. The 16-core achieves 7 GFLOPs and the 32-core achieves 13.4 GFLOPs.

Table I. Results for matrix multiplication

	Core	Arch. 16-cores	Arch. 32-cores
LUTs	1,364	24,390	46,576
DSPs	4	71	135
BRAMs	8/4	140	140
Freq. (MHz)	250	250	250
Cycles	—	77,772,668	39,796,887
Time (s)	—	0.31	0.16
GFLOPs	—	7	13.5
Peak GFLOPs	0.5	8	16
Efficiency	—	86%	84%

Compared to previous implementations, ours has about half of the performance of the dedicated architecture for matrix multiplication in [24], but consumes only about 25% of the resources. Doubling the number of cores of our architecture would provide an architecture with the same performance, assuming enough memory bandwidth. In terms of efficiency, our architecture is better. We also have higher efficiencies compared to the dedicated many-core proposed in [23].

V-B. Configuration of the Architecture for Sparse Matrix Multiplication

We have parallelized the sparse matrix-vector multiplication algorithm to run in a many-core architecture. In this paper, we briefly describe the parallelization process (See [28] for a detailed explanation).

Sparse matrix-vector multiplication is the mathematical operation given by

$$y = A \times x$$

where matrix A is a sparse matrix, x is the input vector and y the result of the product between A and x. Given a matrix A of size $n \times m$, vector x is necessarily of size $1 \times m$ and vector y of size $n \times 1$.

A matrix is typically stored as a two-dimensional array. Each entry in the array represents an element $A_{i;j}$ of the matrix and is accessed by the two indices *i* and *j*. Conventionally, *i* is the row index, numbered from top to bottom, and *j* is the column index, numbered from left to right. For an $M \times N$ matrix, the amount of memory required to store the matrix in this format is proportional to $M \times N$ (disregarding the fact that the dimensions of the matrix also need to be stored).

In the case of a sparse matrix, substantial memory requirement reductions can be realized by storing only the non-zero entries. Depending on the number and distribution of the non-zero entries, different data structures can be used and yield huge savings in memory when compared to the basic approach. In this work we have used Compressed Sparse Column (CSC). The compressed sparse column format stores an initial sparse $M \times N$ matrix A in column form using three one-dimensional arrays.

Work attribution to cores was made by nonzero indexes. This means that instead of row ranges, single rows were attributed to each processor. This attribution is done in a round-robin fashion.

To show that the work scheduling to processors in the previous row assignment is balanced, tests using a data set of matrices were run and the percentage of nonzeros assigned to each processor was measured. Results indicate that, for a system composed of four processors, the load balancing measured by percentage of the total number of nonzeros is around 25% for each processor, guaranteeing a good work load balance.

In our design all necessary data to perform a sparse matrix-vector multiplication is in external memory. Therefore, the system implemented reads all data from external memory and writes the result back to external memory. The algorithm is scalable to any number of cores.

The *DMA* module is responsible for moving data between the external memory and the cores. The DMA is controlled by micro instructions provided by the ARM processor located in the Processing System through an AXI General Purpose interface. The DMA unit is structured in two independent modules which enable it to process read and write operation simultaneously. Each core is composed of an input buffer, a Fused Multiplier-Adder (FMA) and local memory.

Table II represents the performance results obtained for the sparse matrix-vector hardware implementation working at an operating frequency of 100 MHz with two cores. With the available bandwidth using more processors improves marginally the execution time.

Table II. Performance results of the proposed architecture

Test name	Maragal_2	flower_5_4	BIBD_14_7	LD_pilot87
NNZ	4357	43942	72072	74949
M	555	5226	91	2030
NNZ per Col	[0, 139]	[1, 3]	[21, 21]	[1, 96]
ARM exec (us)	128	1644	2055	2222
HW exec (us)	94	1077	1438	1647
HW/ARM	1,18	1,31	1,43	1,35

Each column corresponds to a different test with different matrix and vector inputs. *NNZ* stands for the number of non-zero elements and *M* is then number of rows in the input matrix. We also specify the number of non-zeros per column (*NNZ per Col*).

We have extrapolated our system to determine its performance for different memory bandwidths and compared to previous works ([26], [29], [30], [31], [32] and [27]). The average efficiencies determined are across different input matrices. Our work, presents superior efficiencies (from 44% to 66% on average) in all cases except when comparing to [27] (90% efficiency) and [26] (80% of efficiency). However, the efficiencies presented in [26] are based on a different algorithmic solution and for very specific matrices and the efficiencies presented in [27] are theoretical without taking into consideration limitations from architectural structures, like memory bandwidth, that cannot be obtained as ideally assumed in the model.

VI. CONCLUSION

A configurable hardware/software architecture for high-performance embedded computing was proposed. The many-core architecture is configurable at system-level. A design flow to automatically generate the hardware/software architecture was also proposed that starts with the configuration of the architecture and ends with the implementation targeting an FPGA.

Previous proposals of many-core architectures for embedded systems are based on general-purpose embedded processors. Compared to our many-core, these systems in general have a better support to run control intensive kernels or threads but are less efficient for data intensive applications in terms of performance and area. This is because our cores are simpler and application optimized, and can also support higher operating frequencies.

We have evaluated the architecture for parallel dense matrix multiplication and sparse matrix-vector multiplication. The results show that the architectures generated achieves performances close to those of state-of-the-art dedicated circuits and performance efficiencies near 90% without requiring hardware expertise to design the many-core architecture.

VII. REFERENCES

[1] "Handel-C Language Reference Manual." http://www.pa.msu.edu/hep/d0/l2/Handel-C/Handel%20C.PDF.

[2] "Impulse CoDeveloper C-to-FPGA Tools." http://www.impulseaccelerated.com/products universal.htm.

[3] I. A. Technologies.

[4] S. Mohl, "The Mitrion-C Programming Language," (Lund, Sweden), Mitrionics Inc., 2005.

[5] C. to Verilog. http://www.c-to-verilog.com/.

[6] Bluespec Tecnologies and Solutions. http://www.bluespec.com/

[7] JHDL. http://www.jhdl.org/

[8] MAXELER Technologies. https://www.maxeler.com/products/software/maxcompiler/

[9] MyHDL from Python to Silicon. http://www.myhdl.org/

[10] S. McGettrick, K. Patel, C. Bleakley, "High Performance Programmable FPGA Overlay for Digital Signal Processing", Lecture Notes in Computer Science, Volume 6578, 2011, pp 375-384.

[11] J. Kingyens and J. Steffan, "The Potential for a GPU-Like Overlay Architecture for FPGAs", International Journal of Reconfigurable Computing, Volume 2011 (2011), Article ID 514581, 15 pages.

[12] D. Capalija and T. S. Abdelrahman, "A Coarse-Grain FPGA Overlay for Executing Data Flow Graphs", Workshop on the Intersections of CARL, 2012.

[13] Mplemenos, G.-G.; Papaefstathiou, I., "MPLEM: An 80-processor FPGA Based Multiprocessor System," 16th International Symposium on Field-Programmable Custom Computing Machines, pp.273-274, 2008.

[14] Zhoukun Wang; Hammami, O., "External DDR2-constrained NOC-based 24-processors MPSOC design and implementation on single FPGA," 3rd International Design and Test Workshop, pp.193-197, 2008.

[15] Arteris, "Open Core Protocol," http://www.ocpip.org.

[16] A. Tumeo, et al., "A Dual-Priority Real-Time Multiprocessor System on FPGA for Automotive Applications," in Design Automation and Test in Europe, pp.1039-1044, 2008.

[17] E. Carara, R. de Oliveira, N. Calazans, and F. Moraes, "HeMPS - a Framework for NoC-based MPSoC Generation," in IEEE International Symposium on Circuits and Systems, pp.1345-1348, 2009.

[18] I. Lebedev, et al., "MARC: A Many-Core Approach to Reconfigurable Computing," International Conference on Reconfigurable Computing and FPGAs, pp.7-12, 2010.

[19] Open Cores, "Plasma Processor," http://opencores.org/project,plasma.

[20] Wilson Maltez, Ana Rita Silva, Horácio Neto, Mário Véstias, "Efficient Implementation of Single-Precision Floating-Point Arithmetic Unit on FPGA", in 24th International Conference on Field Programmable Logic and Applications, pp. 1-4, 2014.

[21] Kondo, M.; Nguyen, S.T.; Hirao, T.; Soga, T.; Sasaki, H.; Inoue, K., "SMYLEref: A reference architecture for manycore-processor SoCs," in Asia and South Pacific Design Automarion Conference, pp.561-564, 2013.

[22] N. Seki, et al. "A Fine Grain Dynamic Sleep Control Scheme in MIPS R3000," IEICE Transactions on Information and Systems, VoI.J93-D, No. 6, pp.920-930, 2010.

[23] J. Cappello and D. Strenski, "A Practical Measure of FPGA Floating Point Acceleration for High Performance Computing", in International Conference on Application-Specific Systems, pp. 160-167,2013.

[24] V. Kumar, S. Joshi, S. Patkar, H. Narayanan,"FPGA Based High Performance Double-Precision Matrix Multiplication", in 22nd International Conference on VLSI Design, pp. 341-346, 2009.

[25] W. Jose, Ana Silva, H. Neto and M. Véstias, "Analysis of Matrix Multiplication on High Density Virtex-7 FPGA", in International Conference on Field Programmable Logic and Applications, pp. 1-4, 2013.

[26] Yan Zhang, Y.H. Shalabi, R. Jain, K.K. Nagar and J.D. Bakos, "FPGA vs. GPU for sparse matrix vector multiply". in International Conference on Field-Programmable Technology, pp. 255- 262, 2009.

[27] R. Dorrance, F. Ren, and D. Markovic, "A Scalable Sparse Matrix-Vector Multiplication Kernel for Energy-Efficient Sparse-Blas on FPGAs", International Conference on FPGA, pp. 161-169, 2014.

[28] João Pinhão, Wilson Maltez, Horácio Neto, Mário Véstias, "Sparse Matrix Multiplication on a Reconfigurable Many-Core Architecture", in Euromicro conference on Digital System Design, 2015.

[29] D. Gregg, C. McSweeney, C. McElroy, F. Connor, S. McGettrick, D. Moloney and D. Geraghty, "FPGA Based Sparse Matrix Vector Multiplication using Commodity DRAM Memory", International Conference on Field Programmable Logic and Applications, pp. 786-791, 2007.

[30] L. Zhuo and V.K. Prasanna, "Sparse Matrix-Vector multiplication on FPGAs," in Proceedings of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 63-74, 2005.

[31] S. Kestur, J.D. Davis and E.S. Chung, "Towards a Universal FPGA Matrix-Vector Multiplication Architecture". in International Symposium on Field-Programmable Custom Computing Machines, pp. 9-16, 2012.

[32] S. Sun, M. Monga, P.H. Jones and J. Zambreno, "An I/O Bandwidth-Sensitive Sparse Matrix-Vector Multiplication Engine on FPGAs". IEEE Transactions on Circuits and Systems I, Regular Papers, vol. 59, no. 1, pp. 113-123, Jan. 2012.

RC3E: Provision and Management of Reconfigurable Hardware Accelerators in a Cloud Environment

Oliver Knodel and Rainer G. Spallek
Department of Computer Science
Technische Universität Dresden
Dresden, Germany
{firstname.lastname}@tu-dresden.de

***Abstract*—Heterogeneous systems consisting of general-purpose processors and different types of hardware accelerators are becoming more and more common in HPC systems. Especially FPGAs provide a promising opportunity to improve both performance and energy efficiency of such systems. Adding FPGAs to clouds or data centers allows easy access to such reconfigurable resources. In this paper we present our cloud service models and cloud hypervisor called RC3E, which integrates virtualized FPGA-based hardware accelerators into a cloud environment. With our hardware and software framework, multiple (virtual) user designs can be executed on a single physical FPGA device. We demonstrate the performance of our approach by implementing up to four virtual user cores on a single device and present future perspectives for FPGAs in cloud-based data environments.**

Keywords*—*Cloud Computing; Field Programmable Gate Arrays; High-Level Synthesis; Virtualization.

I. Introduction

Multi-core and multi-threaded processors were in recent years combined with special dedicated hardware accelerators to improve the performance of applications. It is foreseeable that the future of hardware development lies in more and more massively parallel architectures as used in embedded as well as high performance systems [1]. Especially field-programmable gate arrays (FPGAs) provide an energy-efficient way to achieve high performance by tailoring hardware directly to the application.

Numerous application areas requiring high processing capability employ simple computation cores, data structures and algorithms which are highly suitable for the utilization of FPGAs. In particular the use of reconfigurable hardware to accelerate computationally intensive applications has increased steadily over the last decade [2]. FPGAs provide customized hardware performance and low power consumption which makes them interesting for the field of high performance computing and leads to the new discipline of high performance reconfigurable computing (HPRC) [3].

Due to the growing deployment of reconfigurable hardware as accelerators in HPC systems and data centers, it is necessary to simplify the access to such resources and to increase their usability. One possible solution is provided by the integration of reconfigurable hardware in cloud architectures. Cloud computing is a key technology with the potential to transform the whole information technology industry. The idea is in the end that "using 1,000 servers for one hour costs no more than using one server for 1,000 hours" [4]. Providing reconfigurable hardware in such way can raise its acceptance in many scientific and economic fields by accelerating application.

In contrast to conventional computing resources, the integration of reconfigurable hardware into a cloud infrastructure often proves to be difficult and is currently a topic of research only. This article describes our concept for the flexible integration of FPGAs in a multi-user system, making them available to a broader group of users as a cloud service in a data center. For this purpose, the provision in a distributed multi-user environment and a central administration is necessary [5]. We also introduce our concept of virtual FPGAs (vFPGA), which enables one physical FPGA to host multiple vFPGAs from different users simultaneously. Our approach increases the utilization and efficiency even for small user designs.

A resource management system for physical and especially virtual FPGAs cannot work efficiently without an integrated computing framework providing a virtualization environment. Thus, we implement a fully integrated computing framework, allowing easy access to the FPGA resources through common interfaces on hardware and software level and achieving high throughput communications. The virtualization necessary to provide up to four virtual user designs on a single device is another important feature of our approach. Transferring an application or an algorithm to reconfigurable hardware requires fundamental and profound understanding of the hardware. To reduce the development time of computationally intensive applications, an integration of state-of-the-art high-level synthesis (HLS) tools is necessary and also part of the framework.

The following Section II introduces comparable concepts and related research in the field of reconfigurable hardware in cloud architectures and computing frameworks. Section III gives an overview on cloud service models. Section IV shows the implementation of our resource management system with an overview of our FPGA computing framework. As an example, an algorithm is transferred to the FPGA using HLS in Section V. Section VI concludes and gives an outlook.

II. Related Work

Reconfigurable hardware such as FPGAs can be used in data centers for hardware acceleration of special applications with simple data structures and streams. In these cases the reason for the use of FPGAs lies, in addition to their high processing speed, mainly in the comparatively low energy consumption compared to graphic processing units (GPUs) and common processors

2nd International Workshop on *FPGAs for Software Programmers (FSP 2015)*, London, United Kingdom, September 1, 2015.

(CPUs). For cloud services, it is furthermore possible to use FPGAs for anonymization of user requests [6] and to increase security [7]. The integration of reconfigurable hardware in cloud architectures is shown in [8]. An example for the integration of hardware accelerators in the open source cloud management system OpenStack is shown in [9]. A comparable contribution with stronger focus on the transfer of applications into an FPGA grid for high performance computing is shown in [10]. Both applications focus on a single cloud service model with a background acceleration of applications using FPGAs.

A cloud integration of reconfigurable resources requires the virtualization of the resource FPGA. The VirtualRC [11] uses a uniform hardware/software interface to realize communication on different FPGA platforms. BORPH [12] takes a similar approach with a homogeneous interface for hard- and software. The FPGA paravirtualization pvFPGA [13] uses a profound integration of an FPGA device driver in a Xen virtual machine.

Furthermore, there exists a number of frameworks for FPGA hardware accelerators with different features. These range from simple PCIe implementations which provide memory transfers [14, 15], to complex frameworks that also incorporate the integration of DRAM and allow for dedicated computational cores [16, 17]. The cores can be generated by external high-level synthesis tools. The OpenCPI framework [17] includes hardware-specific modules such as PCIe, Ethernet and DDR memory, in which the user application is embedded as a computing core in a data flow architecture. Also Leap [18] presents an interesting framework with a so-called FPGA operating system as management core and the possibility to transfer an application to the FPGA using HLS.

Another related topic which has been arising in recent years are the so-called remote laboratories. Their idea is it to access FPGAs in a server room from the workplace or even from home. Such systems are mainly used in universities for teaching and research purposes [19, 20]. These concepts offer the opportunity to share lab resources by time multiplexing, and to save lab equipment, space and costs [20]. Such systems are a kind of special FPGA cloud architecture in a university and education environment [21].

In contrast to the systems mentioned, our aim is it to build a system with various cloud service models enabling remote FPGA labs for university education, hardware acceleration for HPC and also background acceleration for data centers with multiple users on the same physical FPGA.

III. Overview and Cloud Service Models

The main component of our system is the hypervisor RC3E introduced in Section IV. It manages the resources and provides access to the FPGA devices. The hypervisor is complemented by our computing framework RC2F realizing the virtual partitioning of the FPGAs. By this, multiple users can share the same physical device to maximize utilization. Both the framework and the interface between user applications on hard- and software are presented in Section IV-D.

The crucial point of an integration of FPGAs into a cloud architecture is it to define possible application areas and service models. Before we describe our cloud architecture's hard- and software level, we discuss service models for FPGAs in a cloud environment. In the following we introduce three key service perspectives and compare them with the definition of service models in cloud computing [22]. Fig. 1 gives an overview of our three models and their user-modifiable components.

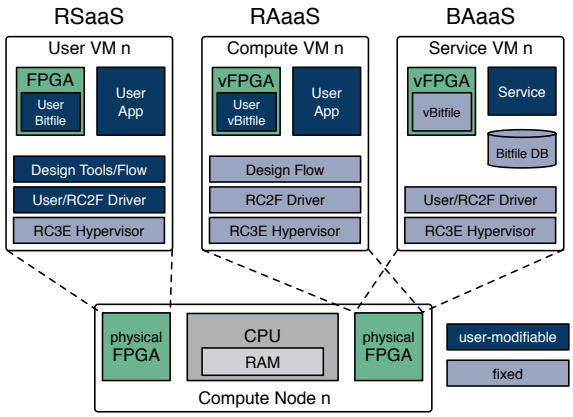


Figure 1: The three service models provided in our cloud environment. In the RSaaS model, users can allocate full physical FPGAs. The RAaaS- and BAaaS model allow multiple concurrent user designs on a single physical FPGA.

A. Reconfigurable Silicon as a Service – RSaaS

Providing full access to the reconfigurable resource, in this model the user can allocate a complete physical FPGA and can implement the hardware of his choice. Allocation and programming are possible with the management framework provided in Section IV. For hardware interface and driver development fully virtual machines with the necessary FPGA devices attached are allocatable by users.

The allocation of vFPGAs is also possible and increases the utilization and efficiency even for small user designs. In this model the whole development flow is provided as a cloud service. The ability to run multiple design flows simultaneously can greatly reduce design exploration time. Parallel to the software flow, the implementation on real hardware including validation and test can be performed on different FPGAs. Since the model allows users to reconfigure the FPGA, it opens new attack vectors that do not exist in current cloud environments. The concept can be compared to the cloud service models Platform as a Service (PaaS) and Infrastructure as a Service (IaaS).

Application areas include for example education and research. It is possible to access FPGAs in a server room from the workplace or even from home, which offers the opportunity to share lab resources and to save lab equipment, space and costs [19, 21, 23].

B. Reconfigurable Accelerators as a Service – RAaaS

Another model with less freedom for the user is the Reconfigurable Accelerators as a Service (RAaaS) model, which is inspired by the HPCaaS concept. In this model the FPGA is used as a simple hardware accelerator and is accessible via the computing framework we introduce in Section IV-D. Only vFPGAs of different sizes are visible, allocatable and useable

by the user. The framework provides a communication API on the host as well as FIFO and memory interfaces on the FPGA. The user only has to design the computation core inside the vFPGA and a host program to send and receive data, which reduces development time and optimizes the design process. Such restrictions furthermore have the advantage that the system is significantly safer than the RSaaS model. The RAaaS model can be compared to the PaaS model.

The concept is suitable especially for research- and development-oriented applications as in this field a limitation of hardware resources can be a bottleneck. Our resource management system provides an integrated batch system for long-running applications without direct user interaction. Moreover, a host program can be submitted to a batch system and program the vFPGA region by itself.

C. Background Acceleration as a Service – BAaaS

Our third model is suitable for applications and services running in common data centers. The vFPGA is not directly visible or accessible by the users. Instead, available applications and services are visible. These services are using vFPGAs in the background to accelerate specific applications. The pre-build bitfiles and host applications are offered by the cloud service provider. Resource allocation and vFPGAs reconfiguration occurs in the background using our resource management system. Because this model provides concrete service applications to the user, it is similar to the Software as a Service (SaaS) model. Application areas include security relevant tasks [6, 7] and in particular computationally intensive routines.

IV. Reconfigurable Common Cloud Computing Environment

The **R**econfigurable **C**ommon **C**loud **C**omputing **E**nvironment – **RC3E** – is our FPGA hypervisor and grants access through a middleware. The system includes resource management and monitoring of FPGA resources. In the following we introduce the FPGA cloud's hardware architecture in Section IV-A, the hypervisor's software architecture in Section IV-B and Section IV-C. Our FPGA computing framework is presented in Section IV-D and the typical design flow is described in Section IV-E.

A. Hardware Infrastructure

Our infrastructure consists of nodes with one processor each and up to two physical FPGA boards. The FPGAs are tightly coupled to the processors using PCIe, nodes are connected to each other via Gigabit Ethernet interconnect. To avoid communication bottlenecks, which may be caused by applications requiring communication between FPGAs, the FPGAs can directly access the global interconnect. The basic structure is a modification of the concept we introduced in [5]. Each physical FPGA can host up to four virtual FPGAs. The system is accessible via a service and management node. Our current architecture consists of two nodes using Xilinx ML605 and VC707 development boards.

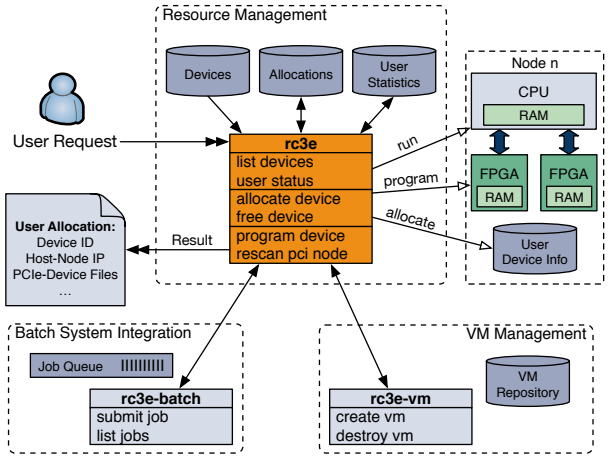


Figure 2: Architecture of the resource management and hypervisor RC3E with additional components (batch system and virtual machines).

B. Hypervisor

In traditional cloud architectures the hypervisor allows users to run their guest operating systems using virtual CPUs. In our approach the hypervisor allows users to implement and execute their own hardware designs on virtual FPGAs (service models RAaaS and BAaaS in Section III). The RSaaS service model additionally allows for an individual operating system with a physical FPGA.

Due to the existence of multiple nodes, physical FPGAs and also vFPGAs, our RC3E hypervisor acts as a resource manager with load distribution. The overall structure of the RC3E hypervisor is shown in Fig. 2. The hypervisor has access to a database containing all physical and virtual FPGA devices in the cloud system and their allocation status. Each device is assigned to its physical host system (node). If each of the three service models are offered in the cloud simultaneously, all FPGA devices have to be assigned to the RAaaS/BaaS model with vFPGAs and the RC2F framework or to the RSaaS model where framework and virtualization are optional.

In the RAaaS/BaaS service model the FPGAs are configured with a basic design containing the RC2F framework, which provides a PCIe endpoint and basic device status information (see Section IV-D). If no vFPGA is allocated and the device is not allocated, most of the clocks in this design are disabled to reduce power consumption. The resource manager always tries to minimize the number of active vFPGAs and to maximize the utilization of physical FPGAs to thereby reduce energy consumption. In the RSaaS service model the user can allocate a complete physical FPGA, which has to be marked separately in the device database (and is therefore excluded from vFPGA allocations).

C. Middleware

The RC3E hypervisor is running on the management node and can access each FPGA node. Users can access the cloud services directly through a middleware with a command line interface on the management node. A client middleware running on a client machine will be added in a future version. It

Table I: Latency of local and remote FPGA status calls and bitstream configuration.

	RC2F Status	Configuration*	PR
Local without RC3E	11 ms	28.370 s	732 ms
Local/Remote Node over RC3E	80 ms	29.513 s	912 ms

* Configuration using JTAG and USB

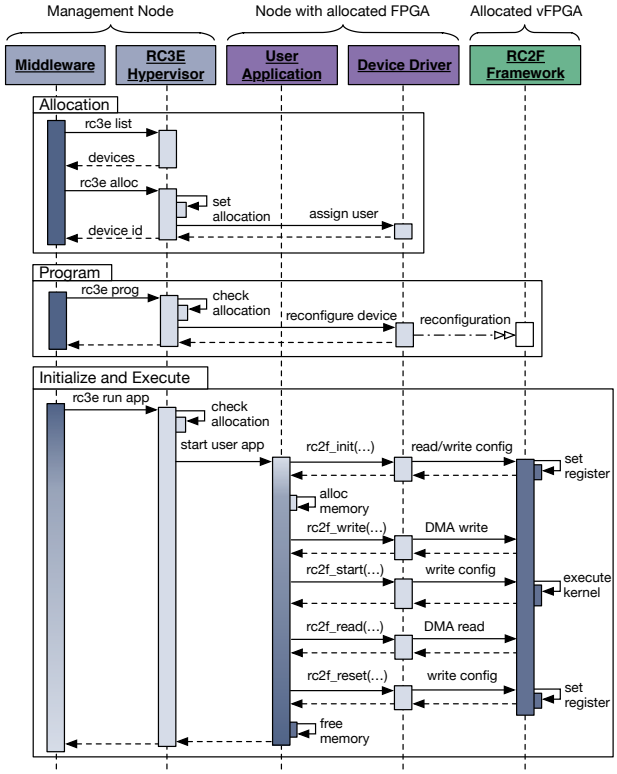


Figure 3: Interaction of middleware, RC3E hypervisor, RC2F user software application and the RC2F framework with the user design on a vFPGA.

is possible to run applications with FPGA acceleration and pre-build bitfiles in the background (BAaaS) without direct allocation of the FPGA resource by the user. The RAaaS model often requires direct interaction, which makes direct allocation of vFPGAs or physical FPGAs with the RC2F framework necessary via the middleware. FPGA configuration and the execution of host applications on the node with the allocated FPGA are possible with separate commands. In both cases, vFPGA configuration is realized by partial reconfiguration (PR).

The RSaaS model allows full access to the allocated FPGA, but allocation, configuration and execution are performed by the middleware. As the hypervisor implements PCIe hot-plugging by restoration of the PCIe link parameters after reconfiguration, the user can also change the PCIe endpoint on the FPGA. Fig. 3 shows the process of resource allocation, programming, initialization and execution. The overhead caused by the RC3E framework is shown in Table I.

In a typical cloud environment there are always sufficient hardware resources to meet user demands. As our academic test

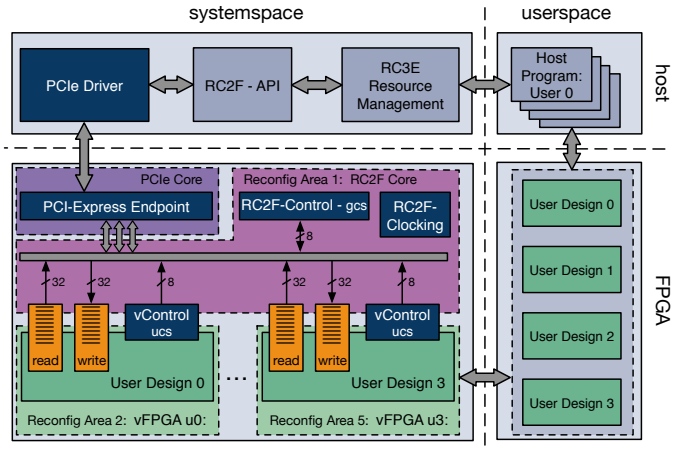


Figure 4: RC2F design with partial reconfiguration areas integrated into a host system.

architecture consists of only two nodes with four FPGAs, we integrated a batch system for long-running applications without direct user interaction to improve overall system utilization. A job of the batch system is to specify the type as well as a configuration file for the FPGAs. Furthermore, we integrated the allocation of user-specific virtual machines with direct access to allocated FPGAs as an extension of the RSaaS service model.

D. Computing Framework – RC2F

Hardware acceleration using FPGAs in a cloud environment requires, in addition to resource management and user administration, a framework realizing the vFPGA concept and allowing integration of user cores. We therefore provide the **R**econfigurable **C**loud **C**omputing **F**ramework – **RC2F** – which is fully integrated into our RC3E environment and provides high communication throughput using PCIe. The framework is typically used in our RAaaS and BAaaS models.

1) Hardware Design: In the RSaaS service model the user has the freedom to implement the hardware of his choice. Such kind of model opens new attack vectors which can cause physical or functional damage to the system. Thus, writing full bitstreams should only be allowed in research (and educational) systems. The preferred basic design for the RAaaS and BAaaS model in our cloud is provided by our RC2F framework and is shown in the lower left part of Fig. 4. The design can provide up to four vFPGAs with independent user designs.

The main part of the RC2F framework consists of a controller managing the configuration and the user cores as well as the monitoring of status information. The controller's memory space is accessible from the host through the API and on the FPGA via dedicated control signals (full reset, user reset, test loopback, etc.). In- and output-FIFO for streaming applications providing high throughput and memory interfaces for configuration are provided as user interfaces. Table II shows the components' resource utilization of the components for implementations with up to four vFPGAs. On a Xilinx Virtex 7 XC7VX485T the resource utilization for a basic design providing four vFPGAs is less than 3%.

Table II: Resource utilization of the individual components for up to four vFPGAs, throughput and memory latency.

Component	LUT	FF	BRAM	Latency	Throughput Core (max)
PCI Endpoint	3,268	3,592	8		
RC2F Control (gcs)	125	255	1	0.198 ms	
1 vFPGA	3,689	3,127	4		
Total	7,082	6,974	13	0.208 ms	≈ 798 MB/s
Utilization*(%)	2.3	1.2	1.3		
2 vFPGAs	4,414	3,790	8		
Total	7,807	7,637	17	0.221 ms	≈ 397 MB/s
Utilization*(%)	2.6	1.3	1.7		
4 vFPGAs	5,139	4,471	16		
Total	8,532	8,318	25	0.273 ms	≈ 196 MB/s
Utilization*(%)	2.8	1.4	2.3		

* Xilinx VC707 evaluation board with a XC7VX485T

2) Communication Interface and Host API: The main communication between FPGA and host is implemented using PCIe. The low-level implementation is based on an IPCore providing simple device files on the host and FIFO as well as memory interfaces on the FPGA [24]. The throughput of the core is limited to 800 MB/s and will thus be replaced in further versions. Fig. 4 gives an overview of the system divided in host/FPGA and user-/systemspace. The figure also shows the FPGA design with PCIe, RC2F core with global configuration space (gcs) and user cores. As interface to the user cores, a user configuration space (ucs) for user-definable commands is implemented as dual port memory. Streaming access is implemented using asynchronous FIFOs, which also divide the system clock from the user clock. Both components together serve as interfaces between the partial reconfiguration areas. The latency for a configuration memory access (gcs in the RC2F module and ucs in the vFPGAs) and the maximal throughput of the FIFOs for concurrent data transfers are shown in Table II.

On the host the FPGA is accessible by PCIe drivers which provide separate device files for each FIFO and each memory. The RC2F host API interacts with the RC3E hypervisor and provides access to the user-allocated resources without a direct user interaction with the device files. For security reasons the device files are protected by access rights. Because of this additional virtualization layer concurrent users can interact with their allocated devices without influencing each other.

The API calls are inspired by the interaction between host and GPU in the NVIDIA CUDA programming environment [25] or the OpenCL [26] framework. The three basic types are (a) global device control, status query and configuration, (b) user kernel control, status query and reconfiguration and (c) data transfers. Due to security reasons only the RSaaS service model allows such interactions.

E. Design Flow

The service models using the RC2F framework are also inspired by the CUDA design flow. Separating computation into hardware and software components, the hardware components will be implemented on the FPGA using Xilinx Vivado HLS and will interact with the software components on the host through our API. The entire design flow is shown in Fig. 5. In addition to the RC2F host-library, a HLS library is necessary

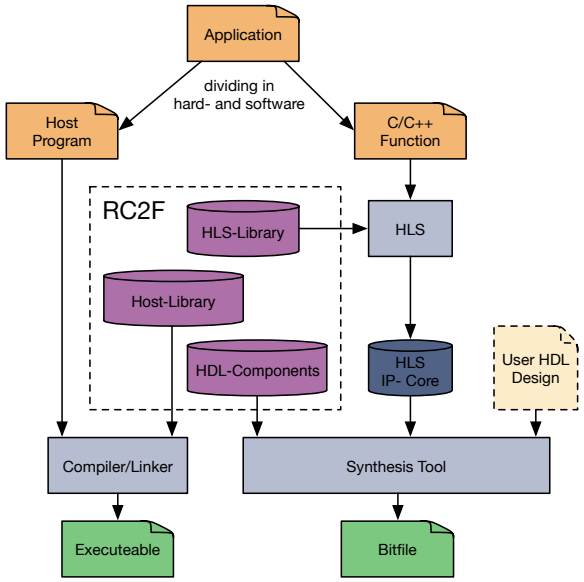


Figure 5: Design flow starting from an application consisting of host program and C-Function as input for HLS.

to provide an integration of the user HLS design into the RC2F hardware and the vFPGAs. HDL components are required for the hardware user interface in the FPGA design. Further extensions of the system will include debugging and tracing of user designs on physical FPGAs.

V. Example Application and Performance Results

In this section an example application is transferred to a vFPGA using HLS and the presented framework. As application we choose a matrix multiplication which offers both high amounts of data and computational complexity. Moreover, we convert the application to work with a streaming-optimized interface inside our RC2F framework. To reach high throughput we stream the data necessary for 100,000 matrix multiplications through the core. Table III gives an overview of the resources necessary for a 16×16 matrix multiplication with up to four user cores and a 32×32 matrix multiplication with up to two user cores.

The host application starts individual parallel user threads sending matrices to the cores, measures runtime and calculates the throughput. For a 16×16 multiplication the throughput of a single core is compute limited by about 509 MByte/s. Two cores on the same physical FPGA share the bandwidth of 800 MByte/s, which results in a communication bottleneck with a throughput of 398 MByte/s per core and in almost the same runtime as a single core. Four simultaneous user cores affect each other significantly stronger, but in the end the overall performance and the utilization of the physical FPGA is much more efficient.

VI. Conclusions and Outlook

This paper presents a way to integrate FPGAs as a resource into a cloud environment and to make them available to multiple users. Three possible cloud service models are introduced

Table III: Matrix multiplication Streaming performance (32 Bit Float) with up to four cores (100,000 multiplications each).

Design	Area				Runtime	Throughput
	LUT	FF	DSP	BRAM	per Core	per Core
16 × 16						
1 vCore	25,298	41,654	80	14	0.73s	509 MByte/s
2 vCores	44,408	76,963	160	19	0.86s	398 MByte/s
4 vCores	81,761	146,974	320	28	1.41s	198 MByte/s
32 × 32						
1 cCore	64,711	125,715	160	14	3.27s	279 MByte/s
2 vCores	123,249	245,103	320	19	3.43s	277 MByte/s

and integrated into our hypervisor RC3E, allowing resource management for virtual FPGA resources using predefined regions on real devices. The resource management is expanded in a subsequent step by a batch system and by the ability to allocate virtual machines.

Furthermore, a framework is presented which aims at simplifying the development of computationally intensive applications on FPGAs. In contrast to other approaches the framework is fully integrated into our hypervisor, which significantly increases virtualization possibilities. On a Xilinx Virtex 7 XC7VX485T the resource utilization for a basic design providing four vFPGAs is less than 3%. The paper concludes with an example application using our framework and showing the performance tradeoff between flexibility of virtualized FPGA resources and a dedicated system.

One of the most important next steps is it to introduce a profound security concept for the system. At the moment the RC3E hypervisor works with access protection and only authorized users can program their allocated device. In the future we plan to implement sanity checking for (partial) bitfiles to avoid both damage by a tampered bitstream and access to the parts not reconfigurable by the users as for example physical ports. Such security aspects are essential for the integration of FPGAs into a productive cloud environment.

In future we plan to improve the hypervisor, the security of the system and we will provide debugging opportunities. Furthermore, we will implement a more profound virtualization of the FPGA devices. Currently, both information about the FPGA type and a free predefined vFPGA region are necessary for design flow. We will try to hide these information from the user and to manipulate the partial configuration file to utilize every feasible vFPGA region. A migration of user designs between vFPGAs and physical FPGAs is also intended.

REFERENCES

[1] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable computing architectures", *Proceedings of the IEEE*, vol. 103, 2015.

[2] T. El-Ghazawi et al., "The promise of high-performance reconfigurable computing", *Computer*, vol. 41, no. 2, 2008.

[3] W. Vanderbauwhede and K. Benkrid, *High-Performance Computing Using FPGAs*. Springer, 2013.

[4] M. Armbrust, A. Fox, R. Griffith, *et al.*, "A view of cloud computing", *Communications of the ACM*, vol. 53, 2010.

[5] O. Knodel, A. Georgi, P. Lehmann, *et al.*, "Integration of a Highly Scalable, Multi-FPGA-Based Hardware Accelerator in Common Cluster Infrastructures", in *Parallel Processing (ICPP), 42nd Int'l Conf. on*, IEEE, 2013.

[6] K. Eguro and R. Venkatesan, "FPGAs for trusted cloud computing", in *Field Programmable Logic and Applications (FPL), 2012 22nd Int'l Conf. on*, IEEE, 2012.

[7] J.-A. Mondol, "Cloud security solutions using FPGA", in *Communications, Computers and Signal Processing (PacRim), 2011 IEEE Pacific Rim Conf. on*, IEEE, 2011, pp. 747–752.

[8] F. Chen, Y. Shan, Y. Zhang, *et al.*, "Enabling FPGAs in the cloud", in *Proceedings of the 11th ACM Conf. on Computing Frontiers*, ACM, 2014, p. 3.

[9] S. Byma, J. G. Steffan, H. Bannazadeh, *et al.*, "FPGAs in the Cloud: Booting Virtualized Hardware Accelerators with OpenStack", *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual Int'l Symposium on*, pp. 109–116, 2014.

[10] J. Dondo Gazzano, F. Sanchez Molina, F. Rincon, *et al.*, "Integrating reconfigurable hardware-based grid for high performance computing", *The Scientific World Journal*, 2015.

[11] R. Kirchgessner, G. Stitt, A. George, *et al.*, "VirtualRC: a virtual FPGA platform for applications and tools portability", in *Proceedings of the ACM/SIGDA Int'l symposium on Field Programmable Gate Arrays*, ACM, 2012.

[12] H. K.-H. So and R. Brodersen, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH", *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 2, p. 14, 2008.

[13] W. Wang, M. Bolic, and J. Parri, "pvFPGA: Accessing an FPGA-based hardware accelerator in a paravirtualized environment.", *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 Int'l Conf. on*, pp. 1–9, 2013.

[14] G. Marcus, W. Gao, A. Kugel, *et al.*, "The MPRACE framework: An open source stack for communication with custom FPGA-based accelerators", in *Programmable Logic (SPL), 2011 VII Southern Conf. on*, IEEE, 2011, pp. 155–160.

[15] M. Jacobsen and R. Kastner, "RIFFA 2.0: A reusable integration framework for FPGA accelerators", in *Field Programmable Logic and Applications (FPL), 23rd Int'l Conf. on*, IEEE, 2013.

[16] K. Vipin, S. Shreejith, D. Gunasekera, *et al.*, "System-level FPGA device driver with high-level synthesis support", in *Field-Programmable Technology (FPT), Int'l Conf. on*, 2013.

[17] J. Kulp, S. Siegel, and J. Miller, "Open Component Portability Infrastructure (OPENCPI)", DTIC Document, Tech. Rep., 2013.

[18] A. Parashar, M. Adler, K. Fleming, *et al.*, "LEAP: A virtual platform architecture for FPGAs", in *The First Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL 2010)*, 2010.

[19] A. Etxebarria, I. J. Oleagordia, and M. Sanchez, "An educational environment for VHDL hardware description language using the WWW and specific workbench", in *Proc. 31st Annual Frontiers in Education Conf.*, vol. 1, IEEE, 2001, T2C–2.

[20] Y. Rajasekhar, W. Kritikos, A. Schmidt, *et al.*, "Teaching FPGA system design via a remote laboratory facility", in *Int'l Conf. Field Programm. Logic and Applications*, 2008, pp. 687–690.

[21] O. Knodel, M. Zabel, P. Lehmann, *et al.*, "Educating hardware design—From boolean equations to massively parallel computing systems", in *Programmable Logic (SPL), 2014 IX Southern Conf. on*, IEEE, 2014, pp. 1–6.

[22] P. Mell and T. Grance, "The NIST definition of cloud computing (draft)", *NIST special publication*, vol. 800, 2011.

[23] H. Zhao and T.-j. Xiao, "An innovative remote experiment system for fpga-based curriculum", in *Proc. IEEE Int'l Symp. IT in Medicine and Education*, 2008, pp. 870–875.

[24] Xillybus Ltd., "An FPGA IP core for easy DMA over PCIe with Windows and Linux", Xillybus Ltd., Haifa, Israel.

[25] D. Kirk *et al.*, "NVIDIA CUDA software and GPU parallel computing architecture", in *ISMM*, vol. 7, 2007, pp. 103–104.

[26] J. E. Stone et al., "OpenCL: A parallel programming standard for heterogeneous computing systems", *Computing in science & engineering*, vol. 12, no. 3, 2010.

Seeing Shapes in Clouds: On the Performance-Cost trade-off for Heterogeneous Infrastructure-as-a-Service

Gordon Inggs, David B. Thomas, George Constantinides
Department of Electrical and Electronic Engineering
Imperial College London
London, United Kingdom
{g.inggs11;d.thomas1;g.constantinides}@imperial.ac.uk

Wayne Luk
Department of Computing
Imperial College London
London, United Kingdom
w.luk@imperial.ac.uk

***Abstract*—In the near future FPGAs will be available by the hour, however this new Infrastructure as a Service (IaaS) usage mode presents both an opportunity and a challenge: The opportunity is that programmers can potentially trade resources for performance on a much larger scale, for much shorter periods of time than before. The challenge is in finding and traversing the trade-off for heterogeneous IaaS that guarantees increased resources result in the greatest possible increased performance. Such a trade-off is Pareto optimal. The Pareto optimal trade-off for clusters of heterogeneous resources can be found by solving multiple, multi-objective optimisation problems, resulting in an optimal allocation of tasks to the available platforms. Solving these optimisation programs can be done using simple heuristic approaches or formal Mixed Integer Linear Programming (MILP) techniques. When pricing 128 financial options using a Monte Carlo algorithm upon a heterogeneous cluster of Multicore CPU, GPU and FPGA platforms, the MILP approach produces a trade-off that is up to 110% faster than a heuristic approach, and over 50% cheaper. These results suggest that high quality performance-resource trade-offs of heterogeneous IaaS are best realised through a formal optimisation approach.**

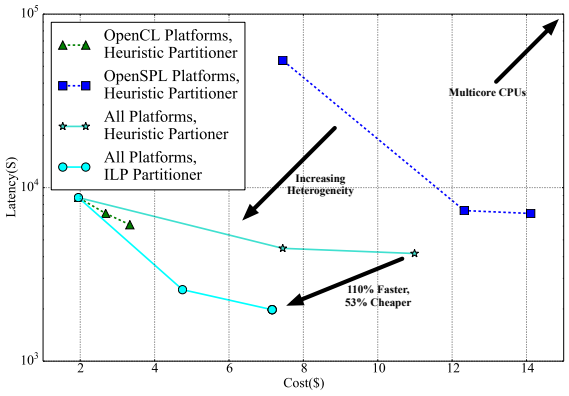


Fig. 1: Latency vs Cost trade-off for 128 option pricing tasks running on 16 heterogeneous Infrastructure-as-a-Service (IaaS) platforms. Full details of the platforms are in Table II.

I. Introduction

Heterogeneous clouds are forming. With the use of FPGA-acceleration in a web-based, commodity application [1], as well as the maturation of heterogeneous computing standards, such as OpenCL and OpenSPL; Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs) are making inroads in High Performance Computing (HPC) data-centres. As a result, providers are mulling Infrastructure-as-a-Service (IaaS) heterogeneous platforms, and it will soon be possible to make use of diverse heterogeneous accelerators without ever having to own any physical hardware. In this paper, we identify and address a central challenge of this new usage mode: partitioning work within a cluster of heterogeneous computing resources. In doing so, we demonstrate that IaaS FPGAs and GPUs can integrate with and enhance Multicore CPUs in the HPC context.

In the past, HPC programmers targeting heterogeneous platforms were limited by the resources that could be traded for performance. The design space that these programmers inhabited was distorted because any implementations were constrained to the handful of devices that capital resources would allow. Heterogeneous IaaS offers the opportunity to interact with a performance-resource trade-off that seamlessly incorporates both capital and operating costs for a much finer time quantum. As opposed to thinking of using a few devices over a period of years, programmers can now target many more devices for only a few hours.

To realise the opportunity of heterogeneous IaaS, significant implementation challenges have to be overcome. Designs are required that efficiently trade device resource utilisation for improved performance for a wide range of heterogeneous hardware targets [2], [3]. Furthermore, programmers now also have to partition their computational workload across multiple designs running on potentially hundreds of heterogeneous devices. We suggest that this is a partitioning problem, similar to selecting the mapping of subtasks to different architectures or partitioned between software and hardware.

Our initial assumption is that many workloads are composed of multiple, atomic (non-communicating) tasks, as evidenced by the popularity of frameworks such as Pig for Apache Hadoop, and algorithms such as Monte Carlo in computational finance. Furthermore, efficient hardware designs can be realised using heterogeneous computing standards and

2nd International Workshop on *FPGAs for Software Programmers (FSP 2015)*, London, United Kingdom, September 1, 2015.

High Level Synthesis tools. In light of these trends, we propose that the partitioning problem for atomic tasks is best addressed using a formal, multi-objective optimisation approach. The trade-off between performance and resource use is realised by varying the allocation of tasks to platforms. The output of this optimisation process is a Pareto optimal trade-off between the total cost of devices used and a measure of performance achieved. A trade-off that is Pareto optimal allows programmers to achieve greater performance in exchange for a higher cost.

In this paper, we show how to achieve a Pareto optimal cost-performance trade-off for multiple atomic tasks upon multiple, heterogeneous IaaS compute devices. In Figure 1 we illustrate our work with the latency-cost Pareto optimal trade-off for a large computational finance computation of 128 option pricing tasks running on 16 heterogeneous IaaS platforms.

Thus, in this paper we:

1) show how the performance-cost design space for IaaS FPGA resources can be formalised into multiple, multi-objective ILP problems.
2) describe how ILP approaches as well as "common-sense"-based heuristics can be applied to solving these optimisation problems, and so generate the trade-off.
3) evaluate our proposed ILP trade-off generation approach against heuristics using a real workload of 128 financial option pricing tasks upon a heterogeneous cluster incorporating 16 CPU, GPU and FPGA-based Platforms from three major IaaS providers.

Our evaluation shows that a heterogeneous set of platforms can significantly outperform its constituent platforms. Furthermore, adopting an ILP approach to partitioning versus a heuristic one achieves a 110% latency improvement and 50% cost improvement in the best case, and performs no worse in the worst case. As the highly performing partitions achieved using the ILP approach illustrate, HPC datacentres of the future should be heterogeneous, and workload partitioning is best done using a formal optimisation approach.

The section that follows provides a brief review of relevant background material on cloud computing usage models, as well as previous work on workload partitioning in distributed computing contexts. We then describe our proposed approach to the partitioning problem: the necessary resource and performance prediction models; formalising the problem as an ILP; outlining our ILP approach for addressing it as well a heuristic approach. We evaluate the partitioning approaches using a workload of financial option pricing tasks upon a heterogeneous cluster of Multicore CPU, GPU and FPGA-based servers. Finally, we conclude and make recommendations for future work.

II. Background

A. Cloud Computing Usage Models

Currently there are two dominant utility or "cloud" computing models: application services, and IaaS [2]–[4]. In the application mode, users pay for access to a service, such as Gmail or SAP, that is provided using computing resources that are hosted in a datacentre owned by the service provider, or that the service provider has leased from a IaaS provider.

TABLE I: Comparison of IaaS offerings. Providers are Microsoft Azure (*MA*), Google Compute Engine (*GCE*) and Amazon Web Services (*AWS*). Prices as of April, 2015.

Provider	Instance Type	Instance Name	Time Quantum (minutes)	Theoretical Peak Performance (GFLOPS)	Rate ($/hour)
MA	CPU	A4	1	416	0.592
GCE	CPU	n1-highcpu-8	10	≈400	0.352
AWS	CPU	c3.4xlarge	60	883	0.924
AWS	GPU	g2.2xlarge	60	2289	0.650

IaaS providers, such as Amazon Web Services (AWS), Google Compute Engine (GCE) or Microsoft Azure (MA), allow for compute resources to be leased directly. These resources are abstracted as virtual servers or instances, accessed via the Internet using protocols such as SSH, which the user may then configure with the desired software. Resources are priced using a rate quoted on a per instance type, per time increment or quantum basis. This rate reflects both the operating and capital expenses of the resources of that instance for the time quantum as well as the provider's profit margin.

The key characteristics of some of the instances from the most popular IaaS providers are reported in Table I. A key feature of each offering is the length of the time quantum, i.e. the minimum increments of time for which the user will be charged. We also observe that the rate reflects performance within the CPU category, hence an instance with twice the peak compute capability of another will roughly cost twice as much.

However, a further observation is that *between* heterogeneous device categories, such as CPUs and GPUs, the link between pricing and performance does not hold. For example the AWS GPU instance listed theoretically offers an extremely attractive performance to cost ratio relative to traditional CPUs, but is priced in the middle of the CPU price range.

B. Workload Partitioning for Heterogeneous Computing

The problem of partitioning computational tasks across distributed, heterogeneous computing resources has been widely studied. The general scenario often considered in the literature is a set of atomic tasks being partitioned across multiple platforms of different capabilities [5], [6]. In this scenario, it is assumed that if a task is allocated to a resource, it will fully occupy that resource until completed. The partitioning is also performed statically, in advance of the tasks' execution, using estimates of performance metrics.

More recent work has considered dynamic allocation during task runtime [2], [3], however this effectively takes the form of static allocation performed on a regular interval with updated task information.

In the atomic task allocation scenario, the general objective is to optimise a measure of performance, often the workload latency or makespan. The *makespan* is the latency between when the first task is initiated until the last result returned. As the tasks are being evaluated on multiple platforms, the

makespan is equivalent to the latency of the platform that takes the longest to complete its assigned tasks.

Two of the suggested approaches in the literature to the performance optimisation problems: *Naive Heuristics* [2], [5] - a simple algorithm is applied to allocate tasks to the available resources. The quality of the partitions produced are highly dependent on the particular tasks and platforms concerned. *Integer Linear Programming* [6] - the partitioning problem is formulated as an optimisation program which can then be solved using ILP techniques, such as the branch and bound algorithms as well as multiple heuristics.

Generally heuristic approaches have been the most studied. Braun's comprehensive study [5] found that simpler heuristics achieve better results than more complex ones. We suggest that this indicates that the truly optimal approach is case-specific, dependent upon subtle dynamics between the task and platforms concerned. ILP appears to be an understudied approach, usually applied only in environments of pressing resource constraint [6]. This lack of attention is likely due to the NP-hard complexity of ILPs in general, and NP-complete in the binary case, prompting concerns over the uncertainty of the time spent finding a solution.

III. Our Partitioning Approach

In this section we describe our approach to partitioning workloads of atomic tasks across heterogeneous IaaS resources so as to achieve a Pareto optimal trade-off between resource use and performance. Throughout our explanation, we use the example of Monte Carlo algorithm-based, financial option pricing tasks.

A. Latency and Cost Models

As described in the previous section, partitioning approaches require some estimation of the critical task characteristics, such as the makespan and financial cost. Hence, models of these characteristics have to be used to predict the performance of the available implementations.

The latency and cost models that we use in Monte Carlo option pricing tasks are given in Equation 1.

$$L(N) = \beta N + \gamma \tag{1a}$$

$$C(L(N)) = \left\lceil \frac{L(N)}{\rho} \right\rceil \pi \tag{1b}$$

The latency model given in Equation 1a is a linear one, comprised of proportional (βN) and constant (γ) terms. The constant term reflects the overhead in initiating the task on a platform, incorporating time spent in communication, device configuration in the FPGA case, etc. The proportional term grows with the input variable (N), reflecting the growth in the number of operations as the task increases in scale. This model reflects the $O(N)$ complexity of the Monte Carlo algorithm, and would need additional polynomial terms for tasks that are more computationally complex.

To find the values of the latency model coefficients (β and γ), we propose a benchmarking procedure for all of the tasks upon all of the available target devices, using a set of N and latency values, as well as weighted least squares regression to solve for the model parameters, β and γ.

The cost model given in Equation 1b reflects the IaaS model described in the previous section. The task latency is divided by the time quantum (ρ), which is then rounded up. This is then multiplied by the platform rate (π). As this model is expressed in terms of the latency model, it is easily generalised, provided an appropriate latency model is available.

Equation 2 describes how we suggest finding the rate (π) for IaaS FPGAs in the current absence of observable market prices.

$$\begin{aligned} \pi &= \text{DBR} \times \text{RDP} \\ \text{DBR} &= (\text{TCO} + \text{PM})\frac{\rho}{P} \end{aligned} \tag{2}$$

The rate is given by the Device Base Rate (DBR), which is the cost per device in the datacentre for the specified time quantum (ρ), scaled by the Relative Device Performance (RDP), the performance of the device relative to the other devices *of the same type* in the datacentre, as per the precedent observed in the market currently. The DBR is given by the annual total cost of ownership for that device (TCO) plus the profit margin (PM) scaled by the time quantum (ρ) to year (P) ratio, i.e. $\frac{\rho}{P}$. To find the TCO, we suggest using a total cost of ownership model for datacentres, such as the simple model published by the Uptime Institute [7].

B. Latency Minimisation on a Budget

The models in the previous subsection describe a single Monte Carlo task upon a single platform. In this subsection we show how these models can be used to trade between characteristics for a workload of τ tasks upon a cluster of μ platforms.

A task-platform allocation could be binary, of whole tasks to platforms. However this doesn't take advantage that tasks are often composed of parallel subtasks. If the degree of parallelism, such as $\boldsymbol{N}$ in the Monte Carlo case, in the set of tasks is sufficiently large, then such an allocation can be real-valued between 0 and 1, representing the proportions of tasks allocated to different platforms. By allowing this *relaxed* allocation, we can cast the partitioning problem as a financial cost constrained, Mixed ILP makespan optimisation problem. Equation 3 gives our formulation of this problem, with the cost constraint (C_k) and task-platform allocation ($\boldsymbol{A}$).

$$\begin{aligned} \underset{\boldsymbol{A} \in \mathbb{R}_+^{\mu \times \tau}}{\text{minimise}} \quad & F_L(\vec{G}_L(\boldsymbol{A})) \\ \text{subject to} \quad & \sum_{i=1}^{\mu} A_{i,j} = 1 \quad j = 1, 2, \ldots, \tau \\ & F_C(\vec{G}_C(\boldsymbol{A})) \leq C_k \quad C_k \in \mathbb{R}_+ \end{aligned} \tag{3}$$

where:

$$\begin{aligned} F_L(\vec{G}_L(\boldsymbol{A})) &= \max(\vec{G}_L(\boldsymbol{A})) \\ \vec{G}_L(\boldsymbol{A}) &= ((\boldsymbol{\beta} \circ \boldsymbol{N}) \circ \boldsymbol{A} + \boldsymbol{\gamma} \circ \lceil \boldsymbol{A} \rceil) \cdot \mathbf{1} \\ & \qquad \boldsymbol{\beta}, \boldsymbol{\gamma} \in \mathbb{R}_+^{\tau \times \mu}, \boldsymbol{N} \in \mathbb{Z}_+^{\tau \times \mu} \\ F_C(\vec{G}_C(\boldsymbol{A})) &= \mathbf{1}^T \cdot \vec{G}_C(\boldsymbol{A}) \\ \vec{G}_C(\boldsymbol{A}) &= \vec{\pi} \circ \left\lceil \frac{\vec{G}_L(\boldsymbol{A})}{\rho} \right\rceil \quad \vec{\pi} \in \mathbb{R}_+^{\mu} \end{aligned}$$

The Latency and Cost models for the financial option pricing tasks that were given in Equation 1 are captured in what we define as the task reduction functions, $\vec{G}_L(\boldsymbol{A})$ and $\vec{G}_C(\boldsymbol{A})$, which provide the platform latency and cost for a given allocation ($\boldsymbol{A}$). In both task reduction functions, $\circ$ represents the Hadamard or entrywise product of matrices or vectors.

In what we define the platform reduction functions, $F_L(\vec{G}_L(\boldsymbol{A}))$ and $F_C(\vec{G}_C(\boldsymbol{A}))$, the platforms' characteristics are combined to a scalar value. In the latency case, this is the makespan, while for financial cost this is the total IaaS utilisation cost.

Many formal optimisation frameworks such as SCIP [8] accept problems in the form given in equation 3, however they do not support non-linear objective or constraint functions such as the maximum and ceiling functions used in the platform latency ($F_L(\vec{G}_L(\boldsymbol{A}))$) and cost reduction ($\vec{G_C}(\boldsymbol{A})$) functions. We now show how these non-linear functions can be captured in equation 4.

$$
\begin{aligned}
\underset{\boldsymbol{A}\in\mathbb{R}_+^{\mu\times\tau}}{\text{minimise}} \quad & F_L \\
\text{subject to} \quad & \sum_{i=1}^{\mu} A_{i,j} = 1 \quad j = 1,2,\ldots,\tau \\
& \vec{G}_L(\boldsymbol{A}) \leq F_L \\
& A_{i,j} \leq B_{i,j} \\
& \quad \boldsymbol{B} \in \{0,1\}^{\mu\times\tau}, i = 1,2,\ldots,\mu, j = 1,2,\ldots,\tau \\
& \frac{\vec{G}_{L,i}(\boldsymbol{A})}{\rho_i} \leq D_i \quad \vec{\rho} \in \mathbb{Z}_+^{\mu}, \vec{D} \in \mathbb{Z}_+^{\mu}, i = 1,2,\ldots,\mu \\
& F_C(\vec{D}) \leq C_k \quad C_k \in \mathbb{R}_+
\end{aligned}
\tag{4}
$$

where:

$$
\begin{aligned}
\vec{G}_L(\boldsymbol{A}) &= ((\boldsymbol{\beta} \circ \boldsymbol{N}) \circ \boldsymbol{A} + \boldsymbol{\gamma} \circ \boldsymbol{B}) \cdot \mathbf{1} \\
&\quad \boldsymbol{\beta}, \boldsymbol{\gamma} \in \mathbb{R}_+^{\tau\times\mu}, \boldsymbol{N} \in \mathbb{Z}_+^{\tau\times\mu} \\
F_C(\vec{D}) &= \vec{D}^T \cdot \vec{\pi} \quad \vec{\pi} \in \mathbb{R}_+^{\mu}
\end{aligned}
$$

We have transformed the non-linear functions in the partitioning problem into additional dependent variables and constraints. Firstly, an additional real variable (FL) is introduced that is constrained to being greater than all of the individual platform latencies, capturing the maximum function in platform reduction function ($F_L(\boldsymbol{A})$). A binary variable ($\boldsymbol{B}$) greater than or equal to the allocation variable, captures the ceiling function in the latency task reduction function ($\vec{G}_L(\boldsymbol{A})$). Finally an integer variable ($\boldsymbol{D}$) captures the ceiling function in the cost reduction function ($\vec{G}_C(\boldsymbol{A})$).

C. Finding the Latency-Cost Tradeoff

The previous subsection describes how to minimise latency for a single, fixed cost constraint, however we seek a method for finding the resource-performance trade-off. The previously described program can be used to find such a trade-off by using ILP evaluation tools such as SCIP [8], through the ϵ-constraint method as described by Kirlik et al [9]. By contrast, we also describe a heuristic approach to finding different resource-performance trade-off points.

For our example of a latency-cost trade-off, the same procedure for both the ILP and heuristic approaches is given below.

1) Find the upper cost bound (C_U): For the ILP approach this can be found by minimising the latency without the cost constraint, i.e. $F_C(\vec{D}) \leq C_k$, as this will give the maximum cost on the Pareto curve. Heuristically, this can be found by dividing work inversely proportional to the individual makespans of the available platforms.

2) Find the lower cost bound (C_L): For both the ILP and heuristic approaches, the lowest cost possible is found by allocating all the tasks to the single platform that completes all of the tasks as cheaply as possible. This gives the lowest cost on the Pareto curve.

3) Iterate between C_L and C_U: For ILP, as per ϵ-constraint method, run the program outlined in Equation 4 for a set of cost constraints (C_k) spaced evenly between the upper and lower bounds, for the desired degree of granularity. For the heuristic approach, a linear combination of the normalised latency-cost product can be used each platform. As the weighting of the cost is increased, the trade-off should move from C_U to C_L.

IV. Evaluation

We now evaluate the claims that we have made with regards to modelling task-device latency and cost characteristics in advance, the efficiency of different partitioning approaches and finally, the generation of a Pareto trade off for cost and performance.

A. Experimental Setup

1) Tasks: The computational workload that we used is the pricing of 128 financial option pricing tasks using the Monte Carlo algorithm. The algorithm is compute bound, with random generation accounting for the bulk of the computations. In addition to all of the option pricing tasks being independent, the simulations within each task can be computed in parallel, hence these can be split between multiple platforms. The fixed parameters for the pricing task operations were generated from within the values from the Kaiserslautern option pricing benchmark[1]. The number of simulations per Monte Carlo task ($\boldsymbol{N}$) was set so as to achieve an accuracy of $\$0.001$ for each task.

2) Platforms: Table II provides the details of the heterogeneous cluster that we have used. The cluster is largely made up of Maxeler and Altera FPGA accelerator boards that communicate with the host using PCIe. The FPGA platforms were programmed using both the OpenSPL and OpenCL heterogeneous computing standards, and the Maxeler and Altera High Level Synthesis tools. The two CPUs are those provided by MA and GCE, and are programmed using POSIX and GCC, while the GPU is provided by AWS and programmed using OpenCL and the Nvidia SDK. The rate for the FPGA devices was calculated using Equation 2, with the parameters given in Table III, and the RDP weighting for each device calculated using the relative application performance.

[1] http://www.uni-kl.de/en/benchmarking/option-pricing/

The heterogeneous standards deliver portable performance: the same OpenSPL designs delivers similar performance upon the two platform targeted, despite being implemented on FPGAs from different vendors. Similarly, the difference in performance between the OpenCL GPU and FPGA implementations can be explained almost entirely by the difference in clock rate, suggesting performance portability across device architectures.

3) Software Framework: For task implementation, execution and partitioning, we used the Forward Financial Framework[2](F^3), an Open Source, Python-based Financial Application Framework. F^3 allows for financial problems to be expressed using a library of domain specific objects. The problems can then be evaluated on range of distributed, heterogeneous platforms efficiently [10].

To support the partitioning of tasks, we have extended F^3 to partition workloads using the approaches in Section III.C. To support the ILP approach, we used SCIP [8] as a black-box Mixed ILP optimiser, with Equation 4 as the input program.

B. Method

First we verified the models that we used as inputs into our partitioning approaches. To verify the cost model, we applied the same cost methodology to the IaaS offerings from Amazon as well as a hypothetical FPGA datacentre. For the latency model, we measured the relative error of the latency predictions for 10 minutes of benchmarking. We used heuristic and ILP approaches to finding partitions for our computational workload for multiple budgets, including the lower and upper cost bound. Finally, we used the partitioning approaches to generate latency and cost curves using the model data as inputs. We then ran the resulting partitions on our experimental hardware that make up the curve, verifying the validity of the partitioner outputs.

C. Results and Discussion

1) Cost and Latency Models: Our latency model is verified in Figure 2. The relative error of the latency predicted versus that seen in reality is within 10% for problems many times the size of the benchmarking subset used. As we will show below, this is sufficiently accurate to generate a workload partition.

We have verified our cost model in Table III. We used the Uptime Institute's datacentre cost model updated to 2015 prices, applied to create hypothetical CPU[3], GPU[4] and FPGA[5] IaaS offerings. We have compared the CPU and GPU to AWS's IaaS offering.

The relatively lower capital recovery periods we used for CPUs and GPUs reflect the faster development cycle of these devices as well as the competitive IaaS market. The number of devices given is the number that would fit within the standard datacentre in the Uptime Institute's model.

Both the GPU and CPU rates are very close to those observed in reality, however both are several percent below

[2]https://github.com/Gordonei/ForwardFinancialFramework
[3]Full CPU model: http://bit.ly/1IdJgNg
[4]Full GPU model: http://bit.ly/1GbKVIT
[5]Full FPGA mode: http://bit.ly/1MKjGmc
[6]http://aws.amazon.com/ec2/pricing/

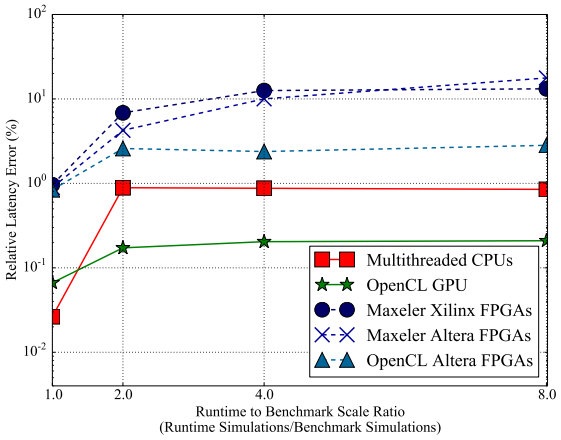


Fig. 2: Latency model prediction error characterisation

TABLE III: Cost model applied to CPUs, GPUs and FPGAs

Parameter	FPGA Model	GPU Model	CPU Model
Device Capital Cost	$5370	$3120	$2530
Energy Use	50W	135W	115W
Number of Devices	5181	5181	5181
Capital Recovery Period	5 years	2 years	2 years
Charged Usage	80%	80%	90%
Profit Margin	20%	20%	20%
Calculated Device Rate	$0.46/hour	$0.64/hour	$0.50/hour
Observed Device Rate[6]	-	$0.65/hour	$0.53/hour

those seen in the market. This is most likely due to an underestimation of the operating costs of the datacentre.

2) Partition Generation: In Table IV, the latency is given for three cost constraints, using the two different partitioning approaches. Both share the same lower cost bound, which is to allocate all of the work to the GPU platform.

The ILP approach demonstrates a significant improvement over the heuristic in the median and upper cost bound values. This difference is explained by the heuristic approach only considering absolute latency and cost, and not taking into account the non-linearities in the latency due to the constant setup time, and in the cost due to the length of time quantum. A good example of this is the CPU platforms, which the heuristic approach does not consider at all, but the ILP does, due to the reduced time quanta both offer.

3) Trade-off Comparison: In Figure 3, we plot the latency-cost design spaces for the two partitioning approaches. For each approach we plot the model data latency-cost trade-off

TABLE IV: Latency-Cost Trade-off for Heuristic and Integer Linear Programming (ILP) Approaches

Cost Level	Metric	Heuristic	ILP	Heuristic / ILP
Cheapest (C_L)	Cost ($)	1.950	1.950	1.0
	Latency (S)	8760.420	8760.420	1.0
Median (C_k)	Cost ($)	7.445	4.749	1.57
	Latency (S)	4468.920	2582.483	1.73
Fastest (C_U)	Cost ($)	10.990	7.160	1.53
	Latency (S)	4172.144	1979.448	2.11

TABLE II: Experimental Heterogeneous Computing Platforms. IaaS providers are Microsoft Azure (*MA*), Google Compute Engine (*GCE*) and Amazon Web Services (*AWS*). Performance was measured using the Kaiserslautern option pricing benchmark.

#	Provider	Device	Programming Standard (Tool)	Lookup Tables	Flipflops	BRAMS	DSPs	Clockrate (Ghz)	Application Performance (GFLOPS)	Rate ($/hour)
4	-	Xilinx Virtex 6 475T	OpenSPL (MaxCompiler 2013.2.2)	298k	595k	1064	2016	0.2	111.978	0.438
8	-	Altera Stratix V GSD8	OpenSPL (MaxCompiler 2013.2.2)	695k	1050k	2567	3926	0.18	112.949	0.442
1	-	Altera Stratix V GSD5	OpenCL (Altera SDK 14.0)	457k	690k	2014	3180	0.25	176.871	0.692
1	AWS	Nvidia Grid GK104	OpenCL (Nvidia SDK 6.0)	-	-	-	-	0.8	556.085	0.650
1	MA	Intel Xeon E5-2660	POSIX (GCC 4.8)	-	-	-	-	2.2	4.160	0.480
1	GCE	Intel Xeon	POSIX (GCC 4.8)	-	-	-	-	2.0	6.022	0.352

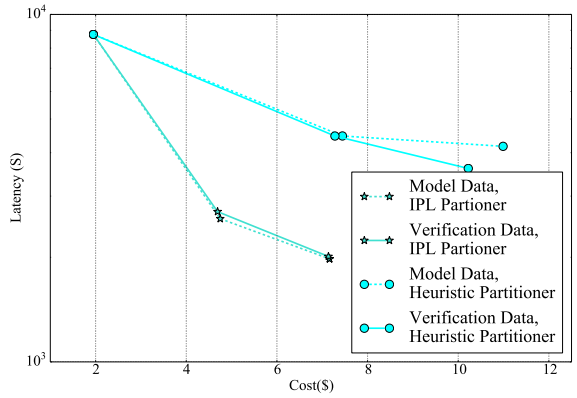


Fig. 3: Partitioner performance model predictions vs measured.

versus the actual trade-off realised when we ran the partitions.

Both approaches' model curves are sufficiently close to the actual data trade-off that a programmer could use these approaches to balance their objectives in advance of actual problem execution. A notable outlier is the upper cost bound of the heuristic approach, where that seen in reality is 12% quicker and 7% cheaper than what is projected by the model. This is consistent with the 10% mean error seen in the latency prediction models.

V. Conclusion

In this paper we addressed the challenge of partitioning workloads across heterogeneous IaaS resources so that lower latencies can be achieved at increased cost. We showed that predictive runtime characteristic models combined with a multi-objective optimisation approach provide an effective methodology for generating Pareto optimal performance-cost trade-offs. We also evaluated two distinct methods for partitioning, showing that our proposed Mixed ILP approach yields a more efficient design spaces than a heuristic one.

Furthemore, our work helps makes the case for heterogeneous IaaS, demonstrating significant performance improvement and cost saving through heterogeneous architectures compared to just using conventional CPUs. However, we argue that these benefits are only realisable if programmers have a means to balance their objectives efficiently, such as our approach to workload partitioning.

In the future we would like to increase the scale of these experiments, both in terms of the number of platforms and tasks, as well as in terms of range of data points explored. There is also significant scope for tuning the partitioners utilised.

Acknowledgements

We are grateful for the funding support from the Oppenheimer Memorial Trust as well as the South African National Research Foundation. We would also like to acknowledge the resources obtained through the Maxeler, Altera and Nallatech University Programs as well an Amazon Web Services Educational grant.

References

[1] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceedings of International Symposium on Computer Architecture*, 2014, pp. 13–24.

[2] E. Oneill, J. McGlone, P. Milligan, and P. Kilpatrick, "SHEPARD: Scheduling on heterogeneous platforms using application resource demands," in *Proceedings of 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, Feb. 2014, pp. 213–217.

[3] J. G. F. Coutinho, O. Pell, E. O'Neill, P. Sanders, J. McGlone, P. Grigoras, W. Luk, and C. Ragusa, "HARNESS project: Managing heterogeneous computing resources for a cloud platform," pp. 324–329, 2014.

[4] J. A. Varia and S. A. Mathew, "Overview of AWS," Amazon, Tech. Rep., 2014.

[5] T. Braun, H. J. Siegel, N. Beck, L. Bölöni, M. Muthucumaru, A. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems," *J. Parallel Distrib. Comput.*, vol. 61, no. 6, pp. 810–837, Jun. 2001.

[6] N. Fisher, J. Anderson, and S. Baruah, "Task Partitioning upon Memory-Constrained Multiprocessors," *11th IEEE Int. Conf. Embed. Real-Time Comput. Syst. Appl.*, pp. 416–421, 2005.

[7] J. Koomey, K. Brill, P. Turner, J. Stanley, and B. Taylor, "A Simple Model for Determining True Total Cost of Ownership for Data Centers," Tech. Rep., 2008.

[8] T. Achterberg, "Scip: Solving constraint integer programs," *Mathematical Programming Computation*, vol. 1, no. 1, pp. 1–41, 2009.

[9] G. Kirlik and S. Sayın, "A new algorithm for generating all nondominated solutions of multiobjective discrete optimization problems," *European Journal of Operational Research*, vol. 232, no. 3, pp. 479–488, 2014.

[10] G. Inggs, D. Thomas, and W. Luk, "A Domain Specific Approach to Heterogeneous Computing: From Availability to Accessibility," in *Proc. First Int. Work. FPGAs Softw. Program. (FSP 2014)*, Aug. 2014.

OpenCL 2.0 for FPGAs using OCLAcc

Franz Richter-Gottfried and Alexander Ditter and Dietmar Fey
Chair of Computer Science 3 (Computer Architecture)
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
91058 Erlangen, Germany
Email: {franz.richter-gottfried, alexander.ditter, dietmar.fey}@fau.de

***Abstract*—Designing hardware is a time-consuming and complex process. Realization of both, embedded and high-performance applications can benefit from a design process on a higher level of abstraction. This helps to reduce development time and allows to iteratively test and optimize the hardware design during development, as common in software development. We present our tool, OCLAcc, which allows the generation of entire FPGA-based hardware accelerators from OpenCL and discuss the major novelties of OpenCL 2.0 and how they can be realized in hardware using OCLAcc.**

FPGA, OpenCL, High level synthesis, High performance computing

I. Introduction

Focusing on embedded systems typically offers two ways to realize the core components: (i) writing software for pre-built devices or (ii) designing custom hardware for the task. Consequently, this decision is usually made very early during development. As it is not easy to find an optimal solution before having designed algorithms and knowing their specific requirements, Hardware/Software Codesign is delaying this decision as long as possible. But still, the individual hardware and software components are developed in mostly disjoint design flows. The most crucial difference between software development and hardware design is the way resources are managed by the developer. While software development assumes a particular hardware to run on and perform best if memory bandwidth and the CPU utilization are saturated, e.g., by exploiting caches or vector instructions; the number of degrees of freedom is even higher when taking the hardware design into consideration, where computational resources and parts of the memory can be tailored to the application's needs.

With the increasing level of parallelism in current processors, language extensions have been developed with the goal to effectively and efficiently distribute work and data on a more abstract and thus, more device independent level to ease development, optimize the solution and increase reusability. These extensions include vendor-specific languages, e.g., CUDA, but also the vendor and device independent OpenCL-standard. Altera was the first FPGA vendor to offer a complete toolchain to implement applications on FPGAs only using OpenCL [1], by deriving an application specific data path form the high level description. Xilinx later also integrated an OpenCL fronted in Vivado HLS and recently presented SDAccel [2]. Though these approaches look very comfortable and promising, performance and efficiency is often limited [3]. In contrast, SOpenCL [4] uses a fixed data-path on the FPGA, which we think leads to a less efficient solution because the application's peculiarities, e.g., the width of data, cannot be exploited. Like the solutions of Altera and Xilinx, our approach derives the FPGA design directly from the algorithm and thus, allows optimal power- and time-efficiency.

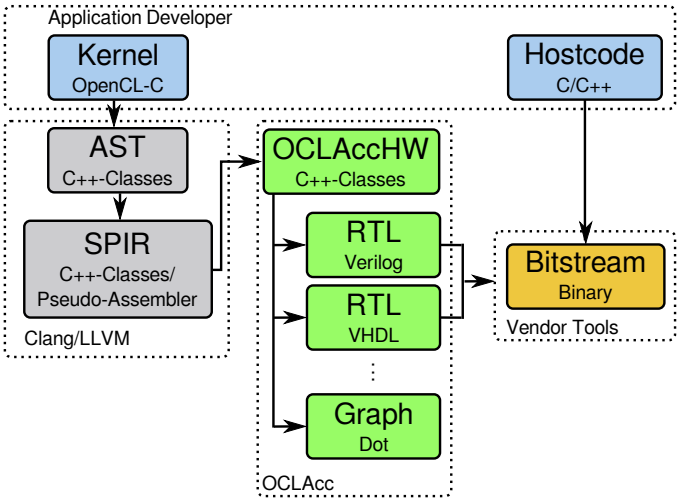


Fig. 1: Overview

In this paper we present OCLAcc [5], an open source generator for configurable logic block based accelerators, and discuss how recent extensions to the OpenCL standard can be applied to hardware design. First, the basic idea of OpenCL is presented in Section II. Section III introduces OCLAcc. New features of OpenCL are discussed in Section IV, followed by the conclusion.

II. OpenCL

OpenCL is a hardware independent framework to separate administration of a program and computation. Administration includes, but is not limited to, coordination of computation, memory allocation or communication, all done by host-code, usually executed by a normal CPU. Computation, expressed as kernel, runs on accelerators like GPUs, but also normal on CPUs. The main advantage of OpenCL is that programmers explicitly parallelize their algorithms. However, portable code does not guarantee portable performance. To fully exploit a target's performance, its properties have to be taken into account, but this is done by methods developers are familiar with, like loop unrolling or latency hiding. For this reason we think OpenCL offers a promising way for software developers to create FPGA-based accelerators.

III. OCLAcc

Figure 1 shows the translation process of OCLAcc. The OpenCL kernel, written in a C-dialect, is translated to Standard Portable Intermediate Representation SPIR by a modified version of Clang, maintained by Khronos. SPIR itself is based on

2nd International Workshop on *FPGAs for Software Programmers (FSP 2015)*, London, United Kingdom, September 1, 2015.

LLVM-IR, so we integrated OCLAcc into LLVM. Translation in OCLAcc happens in two steps. First, SPIR is used to generate OCLAccHW, an internal representation of the data flow, optimized to derive hardware from it. This step is based on basic blocks, which are instruction sequences of maximal length with a single entry and exit in the control flow of the kernel. Inputs and outputs are analyzed to identify streams from and to memory and their static and dynamic indices. The OpenCL standard requires the compiler to provide built-in functions, that can be called by a kernel. They include functions for organization, synchronization and data access, which are mapped to specific components and control inputs. This representation is also used for hardware-specific optimization, like common subexpression identification. HWMap, the second step in OCLAcc depends on the specific hardware used and thus, exploits vendor-tools. By choosing a supported device, the hardware description is derived. Depending on the vendor and the FPGA, OCLAcc either directly instantiates components, generates IP-cores, or relies on inference by the vendor-tools. Scheduling of components is tightly coupled with their generation, because for many parts of the system, parameters like latency or maximum clock frequency are only available when they have been realized. For this reason, clock-driven synchronization of arithmetic units is only done inside of basic blocks, while among blocks, a simple `ready/ack-protocol` is used.

IV. Novelties in OpenCL 2.0

OCLAcc is based on SPIR 2.0 and OpenCL 1.2, though it does not implement the whole standard. This is only necessary to be certified by Khronos and is currently not intended. Instead, we work on features of the recent OpenCL 2.0 standard, of which we expect hardware and developers can benefit from.

Work-group Functions: Until OpenCL 2.0, no built-in functions had been available for data transfer between work-items of the same work-group, e.g., broadcast, reduction. Instead, data transfer via local memory and explicit synchronization had been necessary. In most cases, not all work-items can be executed simultaneously, so work-group functions imply synchronization, and can be handled similar to existing functions like `barrier()`. All work-items have to be processed until every item has reached the built-in. Depending on the function, each work-item may provide data used to compute the result for every item. In hardware, basic blocks write their data to local storage, realized by SRAM on the FPGA, and wait for synchronization. When all work-items of the group have reached that point, the result can be computed and directly used as input for the following block in each item.

Pipes: Pipes realize packet-based communication channels without explicitly managed indices, in contrast to directly allocating a buffer in memory. Created by the host, pipes are passed to kernels as parameters. Applications with several kernels that consume and produce data, can benefit from pipes as they can directly exchange data without manual synchronization. Their FIFO semantic is well known in hardware design, and their realization using SRAM is obvious, but since size and dimension of a pipe are defined by the host, either generic values have to be used by the hardware implementation or the programmer provides constrains, what we expect to be possible in most cases. Pipes also provide an easy way to exchange data with external devices connected to the FPGA or custom logic on the FPGA, without host interaction, e.g., a data stream from an image sensor is pre- and postprocessed and the results are transmitted to the host via PCIe or directly to any other component connected to the FPGA. This makes pipes the most promising extension in OpenCL 2.0, expecially for embedded architectures.

Device-side enqueue: Another new feature is device-side enqueue, which allows work-items to launch kernels without having to return to the host. To realize a dynamically spawned function, a hardware implementation has to be available. OCLAcc has to generate them when the main kernels are translated. The work-queue keeps track of all work groups and items scheduled and has been managed solely by the host, but has now to be accessible by the device itself, which is possible by implementing the queue as FIFO that can also be written to by the kernel.

Shared virtual memory: Global memory on the device has to be seen as storage of memory objects instead of an address space since addresses are not guaranteed to be preserved across kernel instances or between host and device. By introducing shared virtual memory (SVM), host and kernels may exchange pointers to allocated regions. Three kinds of SVM are introduced, with only the first being required by the standard. Coarse-grained sharing works on buffers which can be mapped and unmapped. Pointers to these areas are valid for host and devices. The two kinds of fine-grained SVM allow sharing on basis of individual memory access or even obviate memory handling by OpenCL and allow kernels and host to use memory allocated by `malloc()`. These kernels have the same address space as the host, e.g. when using the CPU as device. Coarse-grained synchronization is similar to memory management before OpenCL 2.0, with the difference that pointers have to be mapped. Finer grained SVM are not to be implemented by OCLAcc in the near future.

V. Conclusion

This paper gives an overview on how OCLAcc derives hardware descriptions from OpenCL, and discusses new features of OpenCL 2.0, which we expect to ease hardware development or extend the possibilities, without having to extend the standard with non-portable specialties.

References

[1] T. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. Singh, "From opencl to high-performance hardware on fpgas," in *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, Aug 2012, pp. 531–534.

[2] "Sdaccel development environment," Xilinx, Tech. Rep. UG1023 (v2015.1), Jun 2015. [Online]. Available: http://www.xilinx.com/publications/prod_mktg/sdx/sdaccel-backgrounder.pdf

[3] D. J. Warne, N. A. Kelson, and R. F. Hayward, "Comparison of high level FPGA hardware design for solving tri-diagonal linear systems," *Procedia Computer Science*, vol. 29, pp. 95 – 101, 2014, 2014 International Conference on Computational Science.

[4] M. Owaida, N. Bellas, K. Daloukas, and C. Antonopoulos, "Synthesis of platform architectures from opencl programs," in *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, May 2011, pp. 186–193.

[5] F. Richter-Gottfried and D. Fey, "OCLAcc: An open-source generator for configurable logic block based accelerators," in *Embedded World Conference Proceedings*, Feb 2014.

Proposal of ROS-compliant FPGA Component for Low-Power Robotic Systems

－case study on image processing application－

Kazushi Yamashina, Takeshi Ohkawa, Kanemitsu Ootsu and Takashi Yokota
Graduate School of Engineering, Utsunomiya University
Yoto 7-1-2, Utsunomiya, Tochigi, Japan, 321-8585
kazushi@virgo.is.utsunomiya-u.ac.jp +81-28-689-6270

Abstract* － *In recent years, robots are required to be autonomous and their robotic software are sophisticated. Robots have a problem of insufficient performance, since it cannot equip with a high-performance microprocessor due to battery-power operation. On the other hand, FPGA devices can accelerate specific functions in a robot system without increasing power consumption by implementing customized circuits. But it is difficult to introduce FPGA devices into a robot due to large development cost of an FPGA circuit compared to software. Therefore, in this study, we propose an FPGA component technology for an easy integration of an FPGA into robots, which is compliant with ROS (Robot Operating System). As a case study, we designed ROS-compliant FPGA component of image labeling using Xilinx Zynq platform. The developed ROS-component FPGA component performs 1.7 times faster compared to the ordinary ROS software component.

Keywords—FPGA; robot; programmable SoC; ROS; component-oriented development;

I. INTRODUCTION

In recent years, the design techniques for building robots are actively studied. In addition, robots (e.g., for disaster relief, unmanned drone, and so on) are required to be autonomous and their robotic software are sophisticated. The robots cannot equip with high-performance processor since they are expected to be in battery operation. On the other hand, FPGA (Field Programmable Gate Array) devices contribute to speed up in fields such like network packet routing and image processing. FPGAs can employ parallel operations for specific functions while software always runs sequentially. But a development of a system using an FPGA is more difficult than that of software since there is a need for implementing them with Hardware Description Language (HDL). High-level synthesis of hardware is also emerging in recent years to implement circuit with C language, however, it is still difficult for ordinary software engineers to handle them.

Robotics depends on various expertise, so they become to be hard to develop [1]. Therefore, the development of systems using FPGA devices needs to reduce cost for easy integration of FPGAs into robots. Component-oriented development is a well-known method for reduction of costs in the development of robotic software. ROS (Robot Operating System) has been proposed as a software platform of component-oriented development of robots [2]. ROS provides a framework of communication layer and a build system for robotic software.

In this study, we propose a hardware component using an FPGA for easy integration of an FPGA into robots, which can be complied and used in ROS system. The ROS-compliant FPGA component contributes to performance improvement of robots using FPGA devices without lowering productivity of robot developers.

This paper presents a practical design of ROS-compliant FPGA component that performs labeling process of images as an example. The component is implemented on a programmable SoC. In addition, the performance of ROS-compliant FPGA component was evaluated by comparing with software.

II. "ROS" FRAMEWORK FOR ROBOT

A. ROS Overview

ROS (Robot Operating System) is released by OSRF (Open Source Robotics Foundation) as an open source project [2]. It is a software platform which provides a framework of communication layer and a build system for robotic software. ROS runs mainly on Linux. Motivation of ROS is to support for *reuse* software and to build robotic systems with software component in robotic research and the development. Figure 1 shows the increase of the number of ROS software packages. The number of packages increases rapidly since released in 2007. In addition, Table 1 shows a number of software packages and citations for several software platforms for robots. It is clear that ROS is becoming a kind of mainstream.

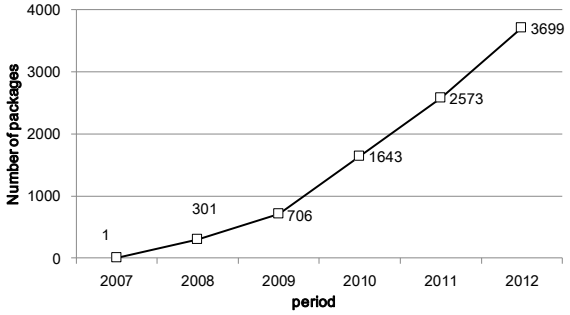


Figure 1 Increase of ROS software packages

2nd International Workshop on *FPGAs for Software Programmers (FSP 2015)*, London, United Kingdom, September 1, 2015.

Table 1 Number of registered packages and citations for each software platform for robots

Platform Name	*Packages*	*Citations**
ROS	3699	4610
RT-middleware [3]	321	1090
OROCOS [4]	Unknown	1540

*google scholar May 5th, 2015

B. ROS communication model

In ROS development, a robot system is designed using a set of component called "*node*" and its communication channel called "*topic*". A robot developer can make a robot system by collecting components from a number of distributed package and connecting them. The developer can also make a *node* when a new function is needed.

The communication model of ROS is based on Publish-Subscribe messaging (Figure 2). Publish-Subscribe messaging is an asynchronous messaging model that ROS Nodes communicate over a topic to each other. The biggest advantage of Publish-Subscribe messaging is a dynamic network configuration since the ROS nodes are bound loosely. Therefore, it is able to add a new ROS node easily.

There are two roles in ROS nodes: *publisher* and *subscriber*. *Topic* is a classification of message as well as a name of the communication path. A *publisher* node publishes a message to a topic. The topic holds a sequence of messages. And any *subscriber* node in the system, which has subscribed to the topic in advance, can receive the message.

Publisher nodes do not know about *subscriber* nodes. In other words, ROS nodes do not have information of communication partner and are bound loosely. Therefore, ROS is able to be dynamic network configuration.

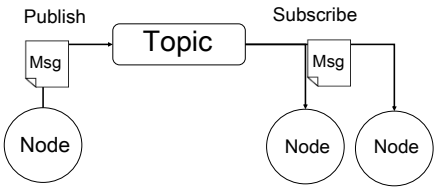


Figure 2 Publish - Subscribe messaging

III. ROS-COMPLIANT FPGA COMPONENT

This section describes the requirements for ROS-compliant FPGA component and its target hardware platform.

A. Requirements of ROS-compliant FPGA component

First, we define "*ROS-compliant"* as follows: An FPGA component is ROS-compliant when the component conforms to publish/subscribe messaging rule so that it can communicate with any other ROS nodes. There are two requirements for ROS-compliant FPGA component.

- The functionality of the ROS-compliant FPGA component is equivalent to that implemented in software.
- The message type and data format used at the input and output of the ROS-compliant FPGA component is equivalent to that implemented in software

An integration of an FPGA into a robotic system needs equivalent functionality to replace a software ROS component with a ROS-compliant FPGA component. Therefore, each ROS message type and data format used in ROS-compliant FPGA component must be same as the software ROS component. ROS-compliant FPGA component aims to improve its processing performance while satisfying their requirements.

B. Structure of ROS-compliant FPGA component

Figure 3 shows the structure of the proposed ROS-compliant FPGA component model. Based on the requirements in the previous section, the component must implement following four functions:

- The encapsulation of FPGA circuits,
- Interface between ROS software and FPGA circuits,
- *Subscribe* interface from a topic, and
- *Publish* interface to a topic.

The FPGA part performs any accelerated processing. There are two roles of software in the component. First, *an interface process for input* subscribes to a topic to receive input data. It is responsible to format the data suitable for the FPGA processing and sends the formatted data to the FPGA. Second, *an interface process for output* receives processing results from the FPGA. It is responsible to format the data suitable for ROS system again, and publishes them to a topic. If other nodes need the data on the topic, the nodes should have subscribed to the topic. Such a structure realizes to make a robot system in which software and hardware cooperate.

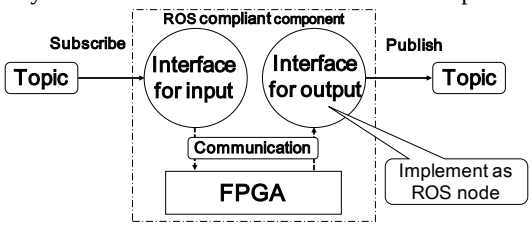


Figure 3 ROS-compliant FPGA component model

C. Implementation on a Programmable SoC

The difference of ROS-compliant FPGA component from a ROS node written in pure software is that processing contains hardware processing of an FPGA or not. Integration of ROS-compliant FPGA component into a ROS system only needs to connect to ROS nodes by Publish/Subscribe messaging in ordinary ROS development style. ROS-compliant FPGA component provides easy integration of an FPGA by wrapping it with software.

We have selected "Programmable SoC" as a target hardware platform for low-power robots. Figure 4 shows assignment of processing on a programmable SoC. SoC (System on Chip) is an LSI that integrates all of the necessary functions to build a system in a single chip. Generally, a hardwired processing logic on an FPGA can achieve better performance at low power consumption compared to software running on a general purpose processor. T. Suzuki et al. exhibited power saving and performance improvement of processing of robotic application using an FPGA [5]. In this study, we use ARM processor and FPGA in programmable SoC. Linux OS and ROS nodes run on the ARM processor, and processing for robot runs on FPGA. Processing which is

suitable for software is assigned to software on ARM, and processing which is suitable for hardware is assigned to hardware. The aim of choosing "Programmable SoC" as a target platform is to improve performance by minimizing burden of hardware development while keeping productivity.

IV. CASE STUDY : LABELING COMPONENT

This section describes implementation of image labeling based on ROS-compliant FPGA component model.

A. Labeling overvew

Processing of image labeling that it puts label number each group of white pixels in binary image. Labeling is used to measure area, angle and length of target in many robotic systems. Figure 5 shows an example of labeling result. To produce a correct result, the labeling need two steps, because raster-scan may label one region as a split several regions. The second step may fix the problem easily. Anyway, this paper describes the first step of labeling, which tends to be time consuming task for software.

B. Hardware implementation overview

Figure 7 shows the block diagram of the hardware. Processing target is full HD image (1920×1080 pixels, about 2M bytes). In this case, labeling operation is applied per line basis, since BRAM of an FPGA is too small to store the entire HD image. In order to exploit maximum efficiency of hardware processing on an FPGA, we designed labeling hardware to be able to label a pixel in a clock.

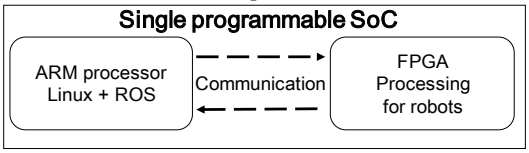


Figure 4 Assignment of tasks on a programmable SoC

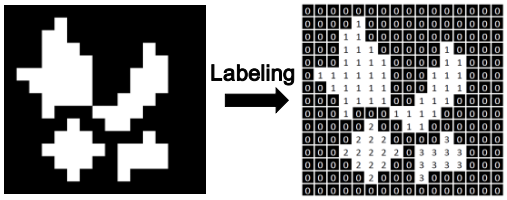


Figure 5 Labeling example

1) Communication between FPGA and ARM processor

In this study, Xillinux [6] is used to communicate between FPGA logic and ARM processor. Xillinux is a platform for Zynq that is released by Xillybus Ltd. Linux (Ubuntu) OS runs on the ARM processor. Xillinux can access to FPGA logic through a specific device file.

Figure 6 shows the communication mechanism of the labelling hardware and the ARM processor. Software can access to FIFOs through device file and read/write data from/to it. The FPGA reads/writes data from/to FIFO by control of read/write enable port at any time.

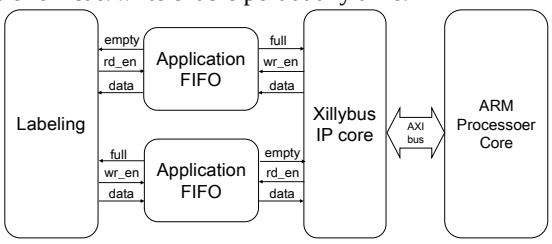


Figure 6 Communication mechanism

2) The role of each module

Table 2 shows the roles of each module in hardware. There are five modules. First, the labeling hardware stores a line of input image to *memory_img*. Immediately after that, *input_controller* inputs pixel data to *label_generator* and l*abel_generator* executes the labeling pixel by pixel. Labeling algorithm needs a pixel data and label numbers of the previous line and previous pixel, so two memories *label_data0* and *label_data1* are prepared. In other words, *label_generator* writes result of current line to *label_data1* while reading from *label_data0*. In the next line, *label_generator* read from *label_data1* and write to *label_data0*.

Detail of "label_generator" is explained in the next section.

Table 2 Function of each module

Module	*Function*
memory_img	Save a line of input image
label_generator	Labeling a pixels
label_data0	Save a line of result (label number)
label_data1	Save a line of result (label number)
FIFO buffer	Buffer for input and output

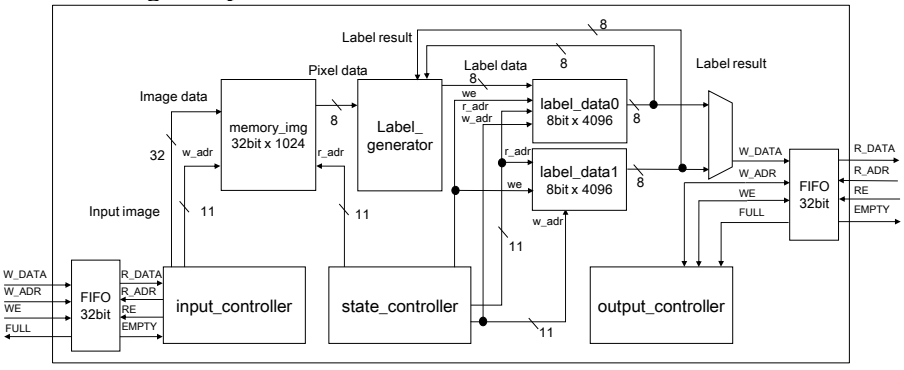


Figure 7 Overall view of hardware

3) Detail of labeling curcuit (label_generator)

We have designed labeling hardware to be able to label a pixel in a clock. Figure 8 shows a circuit diagram of label_generator. There are four 8-bit inputs. One input named "New Pixel" is for pixel data of input image, other inputs are for previous label numbers. In addition, single output named "Output Label" is 8-bit for labeled number.

Whenever "New Pixel" is black ("0"), the output is "0". When it is white, if "*Reference Labels*" are all "0", the circuit outputs an incremented value of previous label number, which is stored in the "*Current Label*" register. On the other hand, if there are any non-zero number in "*Reference Labels*", minimum number of them is output as a result.

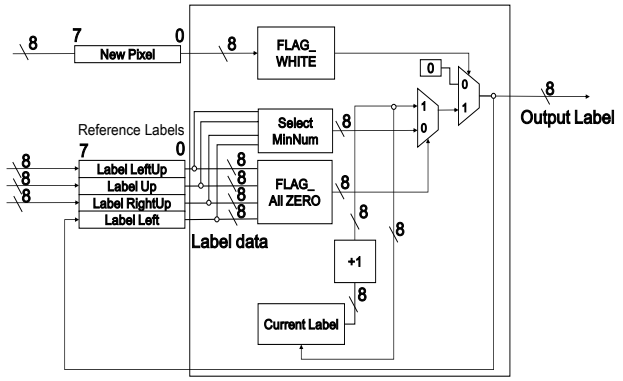


Figure 8 Detailed block diagram of *label_generator*

C. ROS node implementation overview

Figure 9 shows overall view of ROS system which includes ROS-compliant FPGA component. There are four ROS nodes in the system.

- *input_image*: input an image to *data_input (topic)*
- *write2fpga*: receive data from subscribed *data_input* and send it to an FPGA
- *read4fpga*: receive data from an FPGA and publish it to *data_output (topic)*
- *display_result*: receive data form subscribed *data_output* and display it on a console.

Input_image reads input image from a bitmap file and publishes it to topic *data_input*. In *write2fpga*, message received from the subscribed topic *data_input* is sent to FPGA as image data. *Write2fpga* for providing data to the FPGA accesses FIFO through device file and writes data. After labeling on the FPGA, *read4fpga* reads data from the FPGA, similarly. Then the label number data read from the FPGA are published to *data_output* as message. If other node needs labeling results, any node in the system can subscribe to the topic and receive the result from it.

The format of ROS message can be defined at message file like Figure 10 by a developer. A developer can use pre-defined message prepared by ROS, too. In this study, we have defined four fields in the message used in the system.

- *frame_ID*: frame number (32bit integer)
- *width* : width of image (16bit integer)
- *height* : height of image (16bit integer)
- pixels : for pixels data of image (32bit integer array)

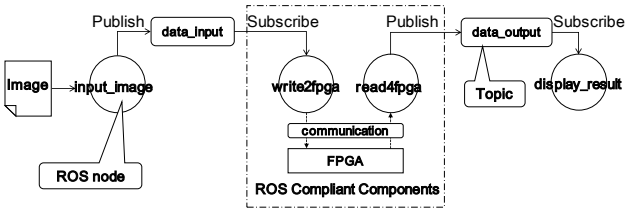


Figure 9 Overall view of ROS system with ROS Compliant FPGA Component

```
int32 frame_ID
int16 width
int16 height
int32[ ] pixels
```

Figure 10 Message file used to define message format

V. PERFORMANCE EVALUATION

This section describes performance evaluation of ROS-compliant FPGA component. The evaluation was conducted in three different conditions.

(1) ROS-compliant FPGA component (ARM + FPGA)
(2) Software only (SW only : ARM)
(3) PC (SW only : PC)

The environment used for (1) and (2) was ZC7Z020 (Xilinx Ltd) on Zedboard. ZC7Z020 is a programmable SoC equipped with ARM Cortex-A9 processor (666MHz) and Artix-7 FPGA on a chip. OS is Ubuntu 12.04 LTS (Xillinux-1.2-eval). In addition, the operating frequency of labeling hardware was set to 100 MHz in (1). The environment of (3) is ordinary PC equipped with Intel Core i7 870 (2.93GHz). OS is Ubuntu 12.10. In addition, Table 3 shows hardware resource utilization of the FPGA.

Figure 11 shows the average of measured processing time in labeling. Resolution of input image is 1920x1080, and the measurement was done using gettimeofday() standard C library function in the software and repeated 10 times. In the environment of (1), the average was 32 ms per frame (*including communication time between the ARM processor and FPGA*), the min/max were 28 ms / 35 ms, respectively. The processing time of ROS-compliant FPGA component was 26 times faster than that of software with the ARM processor, and the ROS-compliant FPGA component performs even 2.3 times faster than that of PC.

Ideally, the number of clocks is 1,920 clocks for the processing per line. In the current implementation, the period is 2,400 clocks because 5 clocks is elapsed to process 4 pixels.

Table 3 Hardware resource utilization (Zynq-7020)

RESOURCE	*UTILIZATION*
Slice Registers	4,123/106,400 (3%)
Slice LUTs	4,114/53,200 (7%)
RAM B36E1	3/140 (2%)
RAM B18E1	11/280 (3%)

Therefore, processing per frame requires 2.6M clocks for 1,080 lines, and the processing time per frame in the FPGA is 26ms (the clock period is 10 ns). This means overhead of the processing time (32 ms) in the experimental system is 6 ms. This is due to communication time between the ARM processor and the FPGA. The ROS-compliant FPGA component performs faster than processing with only software even with care of the communication delay between the software on ARM processor and the FPGA.

Figure 12 shows the total latency from the data input to the data output in ROS-compliant FPGA component and the ROS component with pure software. This latency corresponds to the performance in real situation of robot system. Legends of represents is as follows.

1: Communication of ROS nodes (Publish/Subscribe)
2: From after subscribe to before labeling
3: Labeling
4: From after labeling to before publish
5: ROS node's communication

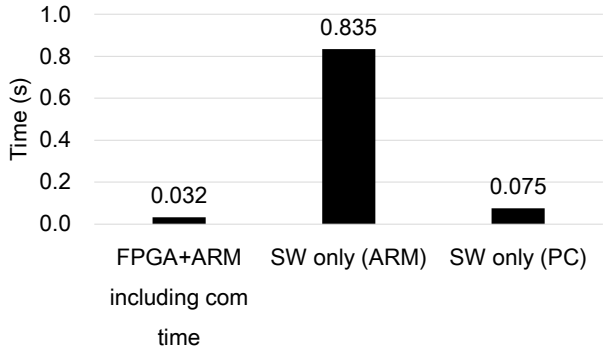


Figure 11 Measured processing time of labeling

In the environment (1), the total latency was 1.99 sec. This is about 1.7 times shorter than that of pure software with the ARM processor. In (1) and (2), communication of ROS nodes occupies a lot of the total latency. In addition, the latency of (3) is much shorter than (1) and (2). Constituent ratio of labeling is about the same in (2) and (3). The time of communication and computation is proportional to processor performance. In (1), the time of computation is reduced drastically, so the time of communication is relatively longer. Operating frequency of ARM processor is 666 MHz in (1) and (2), on the other hand, Intel Core i7 870 is 2.93GHz.

Regarding power consumption, the supply power of labeling hardware on the ROS-compliant FPGA component was estimated with XPower Analyzer by Xilinx that is attached to ISE Design Suite. The total power estimated for our ROS-compliant FPGA component is *0.33W*, the dynamic power is 0.20W, and the static power is 0.13W. Generally, power consumption of high performance processor is about 100W (for example, Intel Core i7 and so on). The power consumption of the proposed ROS-compliant FPGA component is much lower than that of the high-performance processors. Table 4 shows the power consumption of several wheel-based vehicle robots by battery operation. They ranges from 22W to 400W. The estimated power of the labeling component is much lower compared to them. Therefore, ROS-compliant FPGA component can contribute to the performance improvement of robots while keeping low power.

Table 4 Power consumption of robots

ROBOT	BATTELY POWER
Kobuki [7]	22W
iRobot Roomba [8]	30W
Husky A200 [9]	400W

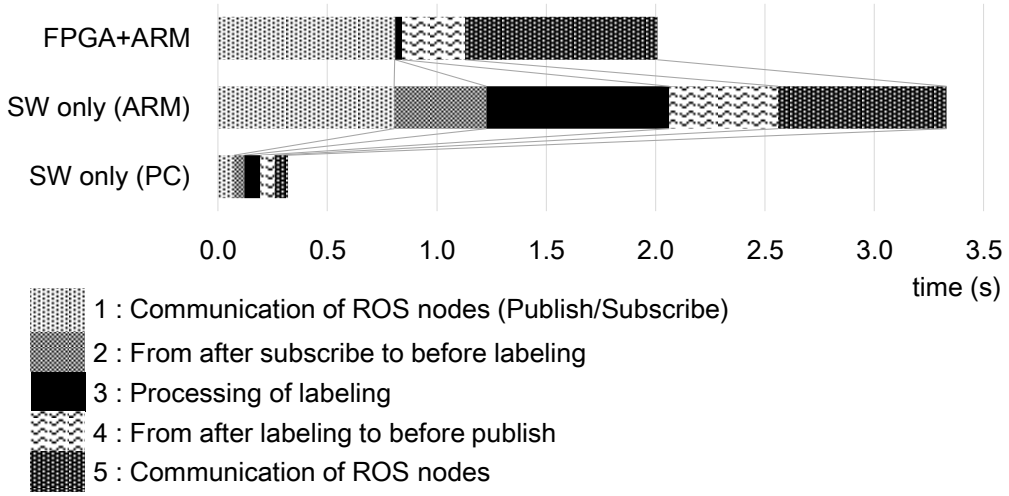


Figure 12 Total latency of the ROS compliant FPGA component

VI. Related works

There are not few papers which report the application of an FPGA onto a robot. This section describes some examples which FPGA is used for robots.

As a digital logic interacting with real-world interface, FPGA is used for robot manipulator with advanced control logic. This is because most of nonlinear controllers need real time mobility operation which is hardly realized by general purpose microprocessors [12] [13].

In previous research [14], autonomous fuzzy behavior control, and sensor-based behaviors are implemented on an FPGA for robot car. These are needed for the human-like driving skills by an autonomous car-like mobile robot.

Another view point of the application of an FPGA onto robotic systems is the design environment of control systems using an FPGA. The development of a system using an FPGA is more difficult than that of software since there is a need for implementing them with Hardware Description Language (HDL). It is still difficult for ordinary software engineers to handle them, so how to reduce the development cost of a system using FPGA is very important. In previous research [16], it is mentioned that the development of a system on an FPGA using traditional programming language: C, C++ and MATLAB improve productivity for robotic developer. Recently, our research group developed an inverted pendulum system using the high level synthesis tool which generates HDL code from pure Java code. The computation time of the control logic of the system is greatly reduced by using FPGA designed with the Java code, without writing any HDL code [17].

FPGA is very effective for robots to reduce computation but connection between software and FPGA is not easy. This paper proposed ROS-compliant FPGA component as a technology which build easy a bridge between software and FPGA. Robotic developer can choose any language to implement robotic software and hardware by ROS-compliant FPGA componentizing processing for robots. Once implementing with the ROS-compliant FPGA component, it can achieve performance improvement of robotic system.

VII. Conclusion

This paper describes ROS-compliant FPGA componentizing of image processing hardware on a programmable SoC. The proposed ROS-compliant FPGA component technology using FPGA devices is aimed to contributes to the easy integration of FPGA into robots.

As a case study, the proposed component with hardwired image labeling on an FPGA is implemented on programmable SoC equipped with the ARM processor and FPGA logic. The ROS-compliant FPGA component on Xilinx Zynq-7020 performs 26 times faster than that of software with the ARM processor, and even 2.3 times faster than that of PC. Moreover, the total latency of the component was 1.7 times faster than that of processing with pure software. Therefore, the ROS-compliant FPGA component achieves remarkable performance improvement, maintaining high development productivity by cooperative processing of hardware and software.

Communication of ROS nodes occupies a lot of execution time in ROS-compliant FPGA component. From now on, another study is necessary for the reduction of ROS node's communication latency.

Acknowledgment

This work was supported by SCOPE (0159-0112), SOUMU, Japan.

References

[1] Robotics Society of Japan (ed.): "Robot technology", Ohmsha, Ltd, 2011.

[2] Open Source Robotics Foundation : http://wiki.ros.org/.

[3] OpenRTM-aist: http://www.openrtm.org/openrtm/ja.

[4] OROCOS: http://www.orocos.org/.

[5] Takuya Suzuki, Yusaku Yamazaki, Hakaru Tamukoh and Masatoshi Sekine : "AMobile Robot System using lntelligentCircuitin Silicon" IEICE Technical Report VLD2011−105 , CPSY2011−68 , RECONF2011−64, pp.83~88, 2012, in Japanese.

[6] Xillybus: http://xillybus.com/.

[7] Yujin Garage: http://garage.yujinrobot.com/

[8] iRobot: https://www.irobot-jp.com/product/comparison.html

[9] Nihon Binary: http://www.nihonbinary.co.jp/index.html

[10] Kazushi Yamashina, Takeshi Ohkawa, Kanemitsu Ootsu, Takashi Yokota : "Fundamental Design of ROS Compliant Components of Image Processing on an FPGA for Robotic Application", conference paper collection of The 77th National Convention of IPSJ , pp.1-169~1-170, 2015, in Japanese.

[11] Kazushi Yamashina, Takeshi Ohkawa, Kanemitsu Ootsu, Takashi Yokota : "ROS Compliant Componentizing of Image processing Hardware on a Programmable SoC", IEICE Technical Report, RESCONF2015-8, pp.41~46, 2015, in Japanese.

[12] Farzin Piltan, N. Sulaiman, M. H. Marhaban, Adel Nowzary and Mostafa Tohidian : "Design of FPGA-based Sliding Mode Controller for Robot Manipulator", International Journal of Robotics and Automation (IJRA), 2, 3, 173 – 194, 2011.

[13] Piltan, F., Rahmani, M., Esmaeili, M., Tayebi, M. A., Cheraghi, M. P. H., Rashidian, M. R., & Khajeh, A. : "Research on FPGA-Based Controller for Nonlinear System", International Journal of U-& E-Service, Science & Technology, Vol.8, No.3, pp.11-28, 2015.

[14] Sánchez-Solano, S., Cabrera, A. J., Baturone, I., Moreno-Velo, F. J., & Brox, M. : "FPGA implementation of embedded fuzzy controllers for robotic applications", Industrial Electronics, IEEE Transactions on, 54(4), 1937-1945.

[15] Tzuu-Hseng S. Li, Shih-Jie Chang and Yi-Xiang Chen : "Implementation of Human-Like Driving Skills by Autonomous Fuzzy Behavior Control on an FPGA-Based Car-Like Mobile Robot", IEEE Trans. on Industrial Electronics, 50, 5 , 867 – 880, 2003.

[16] Leong, Philip Heng Wai, and Kuen Hung Tsoi : "Field Programmable Gate Array technology for robotics applications.", Robotics and Biomimetics (ROBIO), IEEE International Conference on. IEEE, 2005.

[17] Takeshi Ohkawa, Daichi Uetake, Takashi Yokota, Kanemitsu Ootsu, Takanobu Baba : "Reconfigurable and Hardwired ORB Engine on FPGA by Java-to-HDL Synthesizer for Realtime Application,", Proc. 4th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART 2013), pp.45-50, 2013.

Performance monitoring for multicore embedded computing systems on FPGAs

Lesley Shannon*, Eric Matthews*, Nicholas Doyle*, Alexandra Fedorova†
* School of Engineering Science
Simon Fraser University, Burnaby, Canada
{lshannon, ematthew, ndoyle}@ensc.sfu.ca

† Department of Electrical and Computer Engineering
University of British Columbia, Vancouver, Canada
sasha@ece.ubc.ca

***Abstract*—When designing modern embedded computing systems, most software programmers choose to use multicore processors, possibly in combination with general-purpose graphics processing units (GPGPUs) and/or hardware accelerators. They also often use an embedded Linux O/S and run multi-application workloads that may even be multi-threaded. Modern FPGAs are large enough to combine multicore hard/soft processors with multiple hardware accelerators as custom compute units, enabling entire embedded compute systems to be implemented on a single FPGA. Furthermore, the large FPGA vendors also support embedded Linux kernels for both their soft and embedded processors. When combined with high-level synthesis to generate hardware accelerators using a C-to-gates flows, the necessary primitives for a framework that can enable software designers to use FPGAs as their custom compute platform now exist. However, in order to ensure that computing resources are integrated and shared effectively, software developers need to be able to monitor and debug the runtime performance of the applications in their workload. This paper describes ABACUS, a performance-monitoring framework that can be used to debug the execution behaviours and interactions of multi-application workloads on multicore systems. We also discuss how this framework is extensible for use with hardware accelerators in heterogeneous systems.**

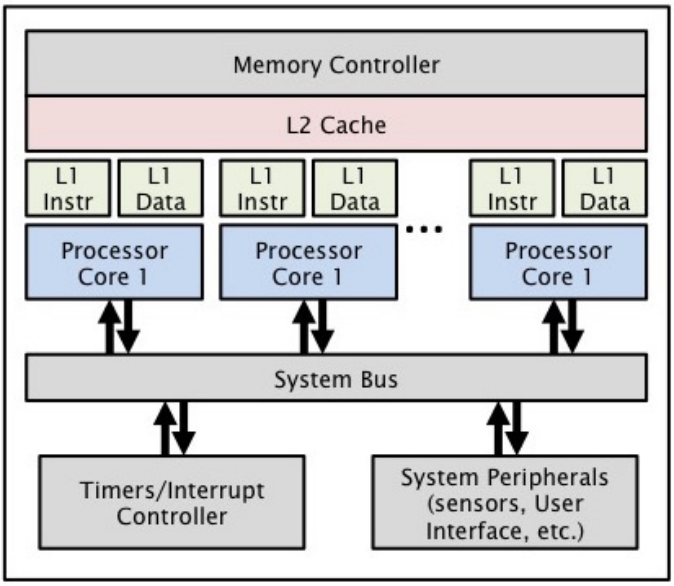


Fig. 1. Typical Multi-Core System Architecture

I. Introduction

The complexity of embedded systems has increased dramatically in the past ten years. Software programmers now commonly design for embedded platforms with multicore processors running operating systems for multi-application workloads that are sometimes even multi-threaded. Recent developments in commercial Field Programmable Gate Array (FPGA) Computer Aided Design (CAD) flows and device architecture suggest that we are approaching the juncture where these software programmers may be able to shift to using FPGAs to implement these types of embedded systems and benefit from the inclusion of hardware accelerators.

Modern Field Programmable Gate Arrays (FPGAs) are large enough to implement hard and soft multicore architectures [1]–[3] in conjunction with custom hardware accelerators. FPGA vendors now also support High-Level Synthesis (HLS), which allows programs written in software languages (e.g. C) to be to be synthesized into actual hardware on a device. Furthermore, vendors provide Graphical User Interfaces (GUIs) that enable software programmers to describe their basic platform architecture and the inclusion of HLS-generated hardware accelerators; they also support embedded Linux kernels for both their hard and soft processors. As such, many of the necessary building blocks for software programmers to use FPGAs to build heterogeneous multicore computing systems already exist.

However, these building blocks unto themselves are insufficient for software programmers to use FPGAs to implement compute platforms at the level of abstraction with which they are comfortable. To enable a seamless transfer from more traditional multicore compute systems to heterogeneous multicore compute systems, software programmers require a complete design and development ecosystem that provides them with the *virtualization* and *visualization* to which they have become accustomed.

In this paper, we concentrate on a software developers need for visualization of an applications behaviour. Programmers need to be able to debug an application's functional behaviour as well as ascertain if (and potentially why) their current

2nd International Workshop on *FPGAs for Software Programmers (FSP 2015)*, London, United Kingdom, September 1, 2015.

solution fails to meet necessary performance requirements. There is significant research ongoing as to how to provide software programmers the necessary functional debug support to develop software that executes as complete custom hardware or as a processor plus one or more hardware accelerators [4]–[8]. The remainder of this paper focuses on performance debug and runtime monitoring.

In a previous paper, we briefly presented a hArdware-Based Analyzer for Characterizing User Software (ABACUS) [9]. It focused on the need for a configurable microarchitectural independent hardware unit that could be used for workload investigation on a single processor core and processor architecture research on FPGAs [9]. This paper describes how the latest work on the ABACUS framework makes it a key visualization component for software programmers to use FPGAs for computing. We explain how ABACUS can be used to provide runtime performance information allowing a programmer to understand why a workload's execution is behaving as it is and what may be the source of its failure to meet performance requirements. We discuss its updated architecture and interface to both the user and OS. We also outline how this framework is not only for single core processors, but also is easily extensible to multicore systems, heterogeneous systems, and functional debug.

The remainder of this paper is organized as follows. Section II motivates why this type of framework is important for software programmers, particularly in heterogeneous systems and then highlights the requirements for such a framework to be accessible to software developers. Section III talks about the challenges and opportunities for performance visualization for software programmers using FPGAs as computing platforms. Related work is summarized in Section IV. We discuss our ABACUS framework in Section V, describing the architecture and system software as well as examples of different types of monitoring units. In Section VI, we outline how ABACUS is designed to be extensible to multicore and heterogeneous systems and can be used for functional debug. Section VII concludes the paper and recommends potential future work.

II. Software Programmer Requirements for Performance Debug

When systems have multi-application workloads, the desire to share a single, coherent memory architecture often results in side effects. An obvious example for a single processor core system is the time needed for context switches to clear the state from the previously executing application and restore the state of the next application to execute. However, consider the case of a coherent multicore system, as shown in Figure 1. In this case, since applications on different processors may need to access memory at any given time – sometimes at the same time – these side effects may become more pronounced, while at the same time less predictable. These delays could negatively impact execution time it may not be able to fetch, process and store/display data in a timely fashion. As such, while the software may be functionally correct (i.e. performing the correct operations), it will not return meaningful results and fail to perform as required.

These types of problems often only arise after a period of execution as they result from software interactions from the various tasks executing on the system. This makes them much easier to detect and understand on the actual execution platform as opposed to in a simulator. However, for this to occur, software programmers must be provided the necessary infrastructure to monitor their system at runtime on the actual platform. First, the ideal infrastructure for this type of monitoring does not require any annotations of the actual software executable. This is because this annotation results in additional execution overhead, resulting in additional side effects. In some cases, this may exacerbate the existing problems. However, in other cases, it may cause new problems that do not exist in the actual system at runtime – or worse, the act of observing the system may artificially "correct" the performance bug, making it now undetectable. This is akin to someone having a bug in their program and then compiling it with the debug flags enabled, which somehow corrects the bug, making it extremely hard to find.

Ideally, this type of performance infrastructure will also have minimal impacts on the memory hierarchy when data is being stored for offline analysis. Depending on the nature and volume of data being stored about an application, it may need to be stored off-chip in the system's main memory. Obviously, this may also result in side effects in program execution as it may potentially introduce new contention to the system's memory arbitration.

For this type of performance monitoring and debug infrastructure to be usable to software developers on an FPGA, they must be able to configure and obtain the data via software *through the on-chip OS* while the FPGA is still configured with their hardware system. Ideally, the system monitoring units should be able to be reconfigured and restarted without having to re-tune the hardware or re-download the hardware system design. Instead, assuming the monitoring units are included as part of the actual system design, programmers can simply alter and update their software, scheduling assignments, etc. to alter system performance – without having to incur costly hardware redesign/re-synthesis time unless absolutely needed.

Additionally, the data obtained at runtime should be sorted and stored per process, if not per thread, so that the developer can track which specific applications (threads) are being penalized by the current resource/scheduling allocation so that this can be corrected more quickly. This type of data becomes even more crucial in heterogeneous systems where software designers may have a hardware accelerator that can be shared amongst a family of applications (e.g. a Discrete Cosine Transform for image processing applications). If both a software and hardware version of the accelerator exist, only a subset of the applications may actually require use of the hardware version to meet timing requirements.

III. Challenges and Opportunities for Performance Visualization on FPGAs

Software programmers choosing to implement their designs on FPGA-based processor systems face some unique challenges; however, they are also provided with some unique opportunities. High-performance processors have long supported hardware counters for performance monitoring, and provide well-developed APIs to use these counters. Soft FPGA processors do not include hardware counters, however, the

latest generation of FPGAs include embedded hard processors that do have hardware counters [1]. Unfortunately, hardware counters are often limited in number ($<< 100$), bit-width, and functionality, and can only be accessed when the processor is not executing application software.

In an FPGA-based system, however, it is possible to build an independent performance monitoring framework using some of the reconfigurable logic. This provides the opportunity for microarchitectural-independent data. Depending on the nature of the soft or hard processor being monitored, the user may be limited as to what signals can be monitored, but snooping a combination of the debug, interrupt, and memory access signals will provide software designers with much of the same information to which they have become accustomed.

In fact, by using the FPGA reconfigurable logic to build monitoring infrastructure, it is completely possible to build new performance monitoring units that are not specifically available. Using this approach, software designers can monitor considerably more complex behaviours than simple counters. Entire Block RAMs (BRAMs) can be allocated to store information such as: histograms of memory access patterns, complete data traces of parts of the programs, stall times and memory latencies, etc. Finally, in an FPGA-based system, the performance monitoring framework need not be integrated into the processor architecture. This allows it to altered/read before, during, or after program execution *and* it does not require any annotation of the software being monitored, reducing the chance for execution side effects.

IV. Related Work

At the accelerator level, there has been some work for visualizing and debugging HLS-generated accelerators. If software designers write their system description in C and then use HLS to generate the circuit, then waveform debugging is not as useful. Instead, a GNU-style debugger enables the programmers to visualize their solution in terms of variables and functions they had originally written as opposed to signals and circuits (and waveforms) created by the CAD flow [4]–[8]. Commercial vendors have also recognized the value of supporting on chip debugging of circuits [10], [11] and HLS designs [12].

Performance Monitoring for multicore systems is an active area of research [13]–[16]. Commercial vendors of multicore systems, such as AMD and Intel, also support profiling frameworks that use the hardware counters embedded in their System architecture [17], [18]. Additionally, previous researchers have also realized the value of using the FPGAs reconfigurable fabric to create additional instrumentation and monitoring circuitry to profile the system in operation, although this has generally been aimed at single core processors [9], [15], [19].

Xilinx has combined these two concepts to provide their SDSoC environment, which uses the embedded hardware counters in the ARM processor in conjunction with performance monitors instantiated in the reconfigurable fabric for monitoring performance on the bus [20]. This is the closest work to our ABACUS framework. However, unlike SDSoC, our performance monitoring framework does not require software to collect its data. Instead, it acts as a completely independent unit, with DMA support, able to write its data back to main memory, to reduce its impact on the actual system. It can also be used to generate interrupts to halt the CPU when specific situations are detected. Unlike SDSoC, ABACUS also has the necessary software to be used in conjunction with the system executing on the FPGA, communicating as needed with its OS. By reserving an OS page in the FPGA-based platform, when the data is uploaded to the OS page from ABACUS, the programmer or the OS can use and aggregate the data to make appropriate design and/or scheduling decisions. A final important feature to our system that is different than SDSoC is its extensibility. It is designed to enable users to create and/or select performance monitoring units from a library of units so that the performance monitoring infrastructure is best suited to the current application set of a particular workload. This extensibility is key to enabling it to support performance monitoring of heterogeneous systems.

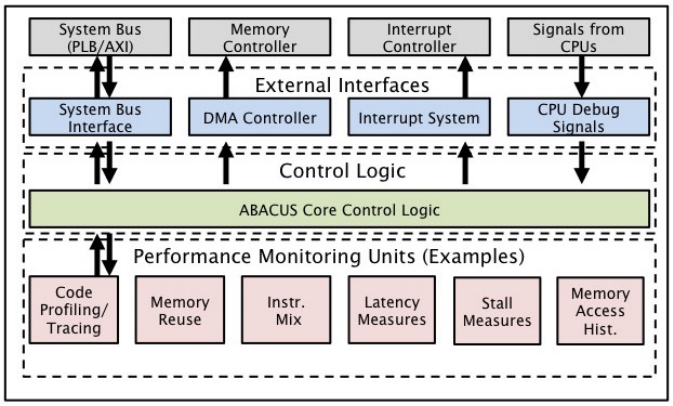


Fig. 2. ABACUS Architecture

V. Our ABACUS Framework

Figure 2 illustrates the basic architecture of our improved ABACUS performance-monitoring unit. Similar to the original design in [9], we still maintain three basic system modules: External Interfaces, Control Logic, and the Performance monitoring units. The current design, however, supports a more complex external interface and more complex Performance Monitoring Units. In the original design, ABACUS was connected to the System Bus and snooped the desired signals from a CPU. Our new version is capable of snooping signals from multiple CPUs and associating the recorded data with a specific process or thread. It also supports DMA and has a corresponding device driver that can be included in the OS kernel. This enables the OS and the user to communicate directly to the device. ABACUS is assigned a specific address range on the system bus and is memory mapped by the OS. This enables status and configuration registers to be read or written through pointer referencing/dereferencing, to enable configuration settings such as reset, enable, and disable for specified processes or physical address ranges. The ABACUS device driver allows access through the ioctl function for reading/writing single registers as well as via mmap to access the full address space of ABACUS. The drivers enable the allocation of a page of kernel memory, enabling ABACUS to use DMA to copy its collected data into software space while the system is running.

The control level now supports the ability to record parallel

data for a specific measurement being run on a multithreaded program executing on multiple processor cores concurrently. It also enables the user to time stamp when specific data was recorded (e.g. memory accesses in a data memory trace). It allows users to trigger ABACUS performance monitoring to start and stop based on various conditions, including the specific number of clock cycles or after a specific memory address access. For example, ABACUS can be triggered to collect user-specified data every time a specific instruction, function, or application executes. It is also able to send an interrupt to the OS to indicate that a specific situation has been detected based on user/OS configuration.

This latest design of ABACUS also includes more complex profiling units to demonstrate the true power and potential of an independent performance monitoring infrastructure not reliant on only hardware counters. For example, we have created a data memory access histogram unit to identify which regions of memory are accessed most frequently. When used in conjunction with our data memory access memory trace unit, this enables programmers to see which data regions are most frequently accessed and in which patterns, potentially facilitating a reorganization of data that reduces cache misses. We have created a memory latency unit to store a histogram of how many cycles it takes for each memory access to be completed. Another complimentary performance monitoring unit is our stall unit that measures the number of clock cycles a processor is stalled, waiting for the completion of an instructions.

This is just a subset of the potential units that can be designed and included as part of ABACUS. However, the key point is that it is easy for the user to select which of these units they wish to instantiate as part of their platform and configure the different parameters associated with each unit. It is also easy for the user to include new units as they are developed. It is possible to boot ABACUS with a set configuration, to start running once the system is powered up. However, the programmer or the OS is free to reconfigure ABACUS at runtime as desired, without having to reboot or re-download the system. Each of the individual performance monitoring units can then be configured independently and activated for any desired subset of the processor cores in the system.

VI. Extending the ABACUS Framework

We are currently extending the ABACUS framework in at least two directions:

Multicore Scheduling: Figure 3 illustrates how ABACUS has been integrated into our multicore PolyBlaze system [3]. Note that the *CPU Debug Signals* illustrated in Figure 2 can be used to monitor *any* signal in the processing system, even those internal to the processor, and their actual connections to the processor are dictated by the types of monitoring units that the user chooses to instantiate. As such, these connections have been excluded Figure 3, with the understanding that ABACUS can connect to *any* signal in a system that is deemed appropriate by the designer.

By including some of the processing units described in the previous section, we have been able to analyze multi-threaded workload execution across the processor cores by aggregating the results of an application's thread execution

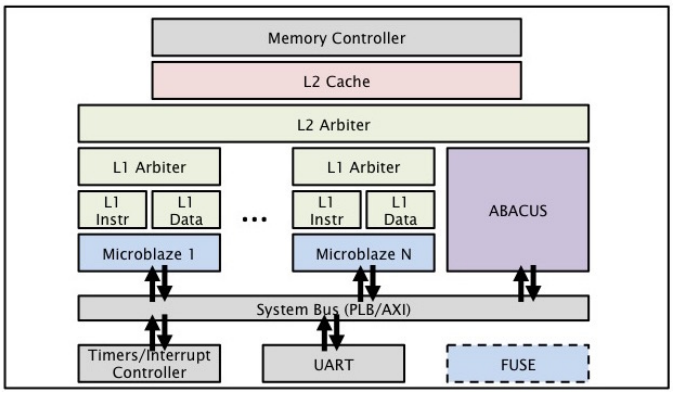


Fig. 3. Integration of ABACUS with PolyBlaze

across the various processors. The key extension we have not yet completed for homogeneous multicore systems is how to use this data to guide the OS scheduler to potentially improve the platforms execution and power efficiency. Although the ABACUS performance monitoring data is currently accessible by the OS, it is not being used to influence scheduling decisions.

Heterogeneous Multicore Systems: The ABACUS hardware framework and software infrastructure are already suitable for incorporation in a heterogeneous multicore system, as shown in Figure 4. The two components currently missing are specific performance monitoring units designed for hardware accelerators and the ability to provide ABACUS' runtime data to the OS to improve the sharing of hardware accelerators. Our current plan is to feed this data into the scheduler in FUSE (shown in Figure 3 in the dashed box), our Front-end USEr API (FUSE) for abstracting hardware accelerators in multicore systems [21]. The API could then use the data to share the hardware accelerators more efficiently and better meet the systems hardware performance requirements.

VII. Conclusions and Future Work

Much of the current research towards making FPGAs accessible platforms for software programmers focuses on HLS and bare metal programming (i.e. no OS) on single processor core systems combined with hardware accelerators. We believe that given the current prevalence of embedded multicore compute systems and the use of operating systems, it is important to consider this next step of development. As FPGA devices increase in size and complexity, with embedded multicore processors supporting operating systems, they become an obvious next generation technology choice for embedded computing systems to facilitate heterogeneity. Providing software programmers with the necessary hardware infrastructure to design these types of compute systems is insufficient to persuade them to adopt these platforms. Instead, we must provide them with the complete design ecosystem to which they have become accustomed, including OS support, debugging and performance monitoring. This paper focused on discussing the needs and opportunities for software designers to create multicore heterogeneous systems. We highlighted how software programmers could use our ABACUS framework in symmetric multicore and heterogeneous multicore systems

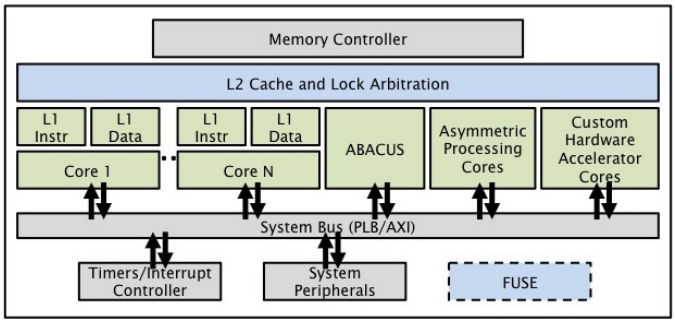


Fig. 4. Integration of ABACUS in a Heterogeneous System

to better understand the execution behaviour of their workloads to make better scheduling and design decisions.

In the future, we will be designing experiments to test ABACUS in heterogeneous compute environments. This will include integrating ABACUS support with a heterogeneous multicore platform virtualization API, such as FUSE [21] and then assessing what different types of performance monitoring units might be most appropriate. Based on this assessment, we hope to generate a basic framework for the key performance monitoring units so that they can be automatically included. The objective is to minimize the number of performance monitoring units software designers would need to generate for their individual systems to further facilitate their design process on a heterogeneous multicore compute platform.

REFERENCES

[1] Xilinx, "All Programmable SoCs and MPSoCs," http://www.xilinx.com/products/silicon-devices/soc.html.

[2] Altera, "Arria 10 ARM-Based SoCs: Reinventing the Midrange," https://www.altera.com/products/soc/portfolio/arria-10-soc/overview.highResolutionDisplay.html.

[3] E. Matthews, L. Shannon, and A. Fedorova, "Polyblaze: From one to many. bringing the microblaze into the multicore era with linux smp support," in *International Conference on Field Programmable Logic and Applications*, August 2012.

[4] J. Goeders and S. J. Wilton, "Effective FPGA debug for high-level synthesis generated circuits," in *International Conference on Field Programmable Logic and Applications*, Sep. 2014, pp. 1–8.

[5] ——, "Using Dynamic Signal-Tracing to Debug Compiler-Optimized HLS Circuits on FPGAs," in *International Symposium on Field-Programmable Custom Computing Machines*, May 2015.

[6] N. Calagar, S. Brown, and J. Anderson, "Source-Level Debugging for FPGA High-Level Synthesis," in *International Conference on Field Programmable Logic and Applications*, Sep. 2014, pp. 1–8.

[7] J. S. Monson and B. Hutchings, "New approaches for in-system debug of behaviorally-synthesized FPGA circuits," in *International Conference on Field Programmable Logic and Applications*, Sep. 2014, pp. 1–6.

[8] ——, "Using Source-Level Transformations to Improve High-Level Synthesis Debug and Validation on FPGAs," in *International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 5–8.

[9] E. Matthews, L. Shannon, and A. Fedorova, "A Configurable Framework for Investigating Workload Execution," in *IEEE International Conference on Field Programmable Technology*, December 2010, pp. 409–412.

[10] Altera, "Quartus II Handbook Version 13.1 Volume 3: Verification; 13. Design Debugging Using the SignalTap II Logic Analyzer," Nov. 2013.

[11] Xilinx, "ChipScope Pro Software and Cores: User Guide," Apr. 2012.

[12] ——, "SDAccel Development Environment: User Guide," May 2105.

[13] N. Ho, P. Kaufmann, and M. Platzner, "A Hardware/Software Infrastructure for Performance Monitoring on LEON3 Multicore Platforms," in *International Conference on Field Programmable Logic and Applications (FPL)*, September 2014.

[14] E. Gibert, R. Martinez, C. Madriles, and J. Codina, "Profiling Support for Runtime Managed Code: Next Generation Performance Monitoring Units," *IEEE Computer Architecture Letters*, vol. 14, no. 1, p. 6265, January 2015.

[15] M. Aldham, J. Anderson, S. Brown, and A. Canis, "Low-cost hardware profiling of run-time and energy in FPGA embedded processors," in *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, September 2011, pp. 61–68.

[16] A. Nair, K. Shankar, and R. Lysecky, "Efficient Hardware-Based Nonintrusive Dynamic Application Profiling," *ACM Transactions on Embedded Computing Systems*, vol. 10, no. 3, pp. 32:1–32:22, April 2011.

[17] AMD, "Performance Profiling without Overhead," http://developer.amd.com/community/blog/2009/07/24/performance-profiling-without-the-overhead/.

[18] Intel, "Process tracing," https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.

[19] L. Shannon and P. Chow, "Using reconfigurability to achieve real-time profiling for hardware/software codesign," in *International Conference on Field-Programmable Gate Arrays*, Feb. 2004, p. 190199.

[20] Xilinx, "SDSoC Development Environment," http://www.xilinx.com/products/design-tools/sdx/sdsoc.html.

[21] A. Ismail and L. Shannon, "FUSE: Front-end User framework for O/S abstraction of hardware accelerators," in *IEEE International Conference on Field-programmable Custom Computing Machines*, May 2011.

Virtualization Architecture for NoC-based Reconfigurable Systems

Chun-Hsian Huang[1,†], Kwuan-Wei Tseng[2], Chih-Cheng Lin[2], Fang-Yu Lin[2], and Pao-Ann Hsiung[2]
[1]Department of Computer Science and Information Engineering, National Taitung University, Taiwan
[2]Deptartment of Computer Science and Information Engineering, National Chung Cheng University, Taiwan
[†]Email: huangch@nttu.edu.tw

I. Introduction

To further enhance the capacity of parallel processing, the Network-on-Chip (NoC) is gradually adopted in a System-on-Chip (SoC) design, instead of the conventional bus architecture. Further, due to the support of partial reconfiguration technology, the Partial Reconfigurable Regions (PRRs) in an FPGA device can be configured as an IP core, such as a General-Purpose Processor (GPP) or a hardware accelerator. As a result, the Processing Elements (PEs) can be dynamically reconfigured on-demand in an NoC-based reconfigurable systems [1]. However, although the partial reconfiguration technology enhances system flexibility so as to meet different application requirements, the resource utilization of hardware logic is still restricted, owing to the limitation of NoC-based infrastructure. This means that, when a software application task is mapped to a PE, this PE is thus blocked and cannot be used by other application tasks, until the previous application finishes. In fact, the PE used is not accessed by the application all the time, which leads to a waste of computing resources.

To solve the above issue, we propose a virtualization architecture for NoC-based reconfigurable systems. The motivation of this work is to develop a service-oriented architecture that includes Partial Reconfigurable Region as a Service (PRRaaS) and Processing Element as a Service (PEaaS) for software applications. According to the requirements of software applications, new PEs can be created on-demand by (re)configuring the logic resource of the PRRs in the FPGA, while the configured PEs can also be virtualized to support multiple application tasks at the same time. As a result, such a two-level virtualization mechanism, including the gate-level virtualization and the PE-level virtualization, enables an SoC to be dynamically adapted to changing application requirements. Therefore, more software applications can be performed, and system performance can be further enhanced.

II. Virtualization Architecture Design

The proposed design is based on a 2D-mesh architecture [2], as shown in Fig. 1. Different from the virtual channel design [3] that focuses on reducing congestion on an NoC, this work further introduces the concept of virtualization. In our current implementation, each PE can be virtualized as two virtual PEs to support two application tasks at the same time.

Besides adopting the partial reconfiguration flow to realize the PRRaaS, a new NI design and a new router design are proposed to realize the PEaaS, as shown in Fig. 2. To enable two different application tasks to access the same PE, in a

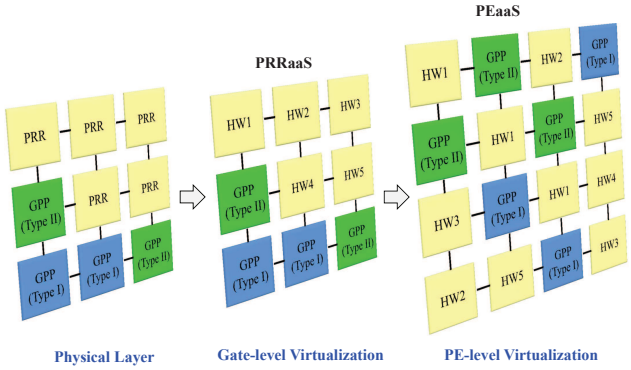


Fig. 1. Virtualization Architecture

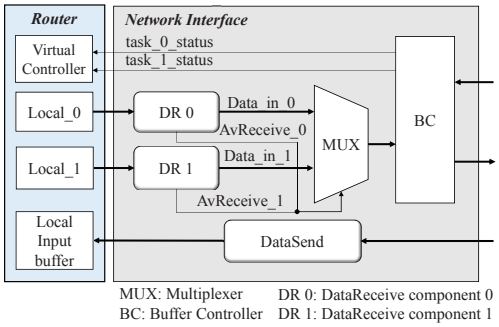


Fig. 2. Network Interface Design

router, two local ports are implemented to individually connect to the two DataReceive components (`DR0` and `DR1`) in the NI for supporting the PE-level virtualization. The DataReceive component is responsible for receiving the flits from a router, and all the received flits are then reconstructed to be a complete packet. When the reconstruction process of the packet finishes, the DataReceive component thus invokes the corresponding signal `AvReceive` (asserted high). Finally, the packet is transferred to the PE through the buffer controller using the signal `Data_in_0` or `Data_in_1`.

To support the PE-level virtualization, each router also includes a virtualization controller that contains two specific signals, namely `task_0_status` and `task_1_status` to control the virtualization mechanism. Initially, the router would act as a conventional one that performs only an application task, in which one of the two local port is disabled. When a

2nd International Workshop on *FPGAs for Software Programmers (FSP 2015)*, London, United Kingdom, September 1, 2015.

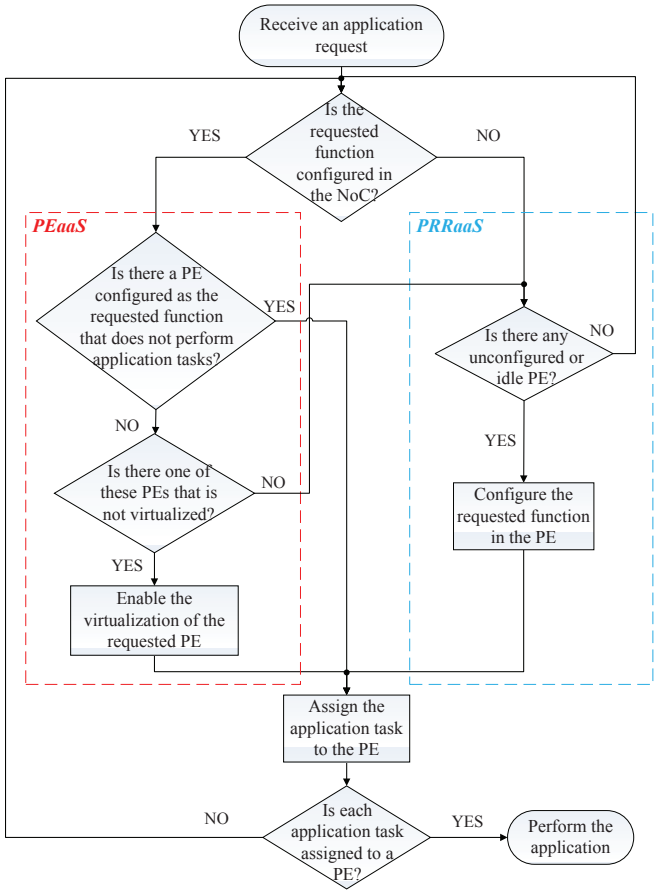


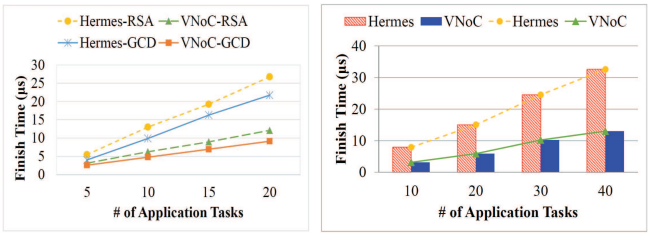
Fig. 3. Adaptation Management

new application task needs to execute on the same PE, PE-level virtualization is then invoked. The unused local port is enabled to receive the packet flits from another application task. As a result, the flits received from two different application tasks can be individually and simultaneously transferred to the two DataReceive components. When the signal `AvReceive_0` or the signal `AvReceive_1` is asserted high, the corresponding signal (`Data_in_0` or `Data_in_1`) is thus used to transfer a complete packet to the PE. Here, based on the first-come-first-served scheduling policy, two different application tasks can be executed on the PE in an interleaving way. As a result, from the viewpoints of software applications, the PE is virtualized as two PEs.

To support the virtualization architecture, an adaptation management mechanism, as illustrated in Fig. 3 is also proposed to receive application requests. This mechanism is realized as a software program executed on a specific PE called global manager. By interfacing with the virtualization architecture, the PRRaaS and PEaaS can be performed for software applications.

III. Experiments

We implemented the virtualization design as a 3×3 mesh NoC architecture on the Xilinx Virtex 6 FPGA. To evaluate the



(a) GCD and RSA are individually virtualized (b) Both GCD and RSA are virtualized at the same time

Fig. 4. System Performance Analysis

proposed design, a conventional Hermes NoC design [2] was also implemented for comparison. Two PEs, including a RSA function and a Greatest Common Divisor (GCD) function, were used to execute multiple application tasks. Compared to the Hermes NoC, supporting PE virtualization needs an extra 1% of slice registers and an extra 2% of slice LUTs. The resource overheads in terms of additional reconfigurable resources are small and acceptable.

To evaluate performance improvement, different numbers of application tasks were applied to both the Hermes NoC and the proposed design (VNoC). The GCD function, the RSA function, and both the GCD and RSA functions were virtualized to support different numbers of application tasks, as shown in Figures 4(a) and 4(b), respectively. We can observe that, when the number of application tasks increases, performance improvement becomes more and more significant. This is because, through the support of the virtualization mechanism, a PE is no longer blocked by only an application task, and it can be used interleavingly by the two application tasks. However, in a conventional NoC, when an application task is mapped to a PE, another application task cannot be mapped to this PE, even though the PE is not used by the application all the time. According to our experimental results, the VNoC can accelerate by 1.5x to 2.5x the processing time required by using the conventional NoC design.

IV. Conclusion and Future Work

This work proposes an NoC-based virtualization design, which also provides the support of PRRaaS and PEaaS for software applications. By using this proposed design, both the utilization of system resources and system performance can be further enhanced. All the hardware adaptation processes are abstracted for the software applications and managed by the global manager, and thus software programmers can focus on the development of applications. In the next phase, we will extend the PE-level virtualization to support at most four software tasks. Further, the machine-learning method will be integrated into the adaptation management mechanism to provide a more intelligent management.

References

[1] S. et al., "Learning-based adaptation to applications and environments in a reconfigurable network-on-chip," in *DATE*, Mar. 2010, pp. 381–386.

[2] M. te al., "Hermes: an infrastructure for low area overhead packet-switching networks on chip," *INTEGRATION, the VLSI Journal*, vol. 38, no. 1, pp. 69–93, Oct. 2004.

[3] K. et al., "Supervised sharing of virtual channels in networks-on-chip," in *SIES*. IEEE, Jun. 2014, pp. 133–140.

A Comparison of High-Level Design Tools for SoC-FPGA on Disparity Map Calculation Example

Shaodong Qin, Mladen Berekovic
Chair for Chip Design for Embedded Computing, Technische Universität Braunschweig
D-38106 Braunschweig, Germany
Email: {qin, berekovic}@c3e.cs.tu-bs.de

***Abstract*—Modern SoC-FPGA that consists of FPGA with embedded ARM cores is being popularized as an embedded vision system platform. However, the design approach of SoC-FPGA applications still follows traditional hardware-software separate workflow, which becomes the barrier of rapid product design and iteration on SoC-FPGA. High-Level Synthesis (HLS) and OpenCL-based system-level design approaches provide programmers the possibility to design SoC-FGPA at system-level with an unified development environment for both hardware and software. To evaluate the feasibility of high-level design approach especially for embedded vision applications, Vivado HLS and Altera SDK for OpenCL, representative and most popular commercial tools in market, are selected as evaluation design tools, disparity map calculation as targeting application. In this paper, hardware accelerators of disparity map calculation are designed with both tools and implemented on Zedboard and SoCKit development board, respectively. Comparisons between design tools are made in aspects of supporting directives, accelerator design process, and generated hardware performance. The results show that both tools can generate efficient hardware for disparity map calculation application with much less developing time. Moreover, we can also state that, more directives (e.g., interface type, array reshape, resource type specification) are supported, but more hardware knowledge is required, in Vivado HLS. In contrast, Altera SDK for OpenCL is relatively easier for software programmers who is new to hardware, but with the price of more resources usage on FPGA for similar hardware accelerator generation.**

I. Introduction

Nowadays, SoC-FPGA, which consists of embedded CPUs (e.g., ARM), programmable logics and a rich set of peripherals, is being popularised as an embedded system platform in robot vision field. Zedboard and SoCKit development board are two typical SoC-FPGA boards supported by Xilinx and Altera, respectively. However, for a typical application targeting for SoC-FPGA platform, the hardware part and software part of the application have to be designed under different development environment with different programming languages, which becomes the barrier of its popularization.

Therefore, high-level design tools, which can design the application at system-level with high-level programming languages, are promoted by FPGA vendors, such as Vivado High-Level Synthesis (HLS) [1] provided by Xilinx, Altera SDK for OpenCL [2], Stratus HLS [3] by Cadence, Synphony C Compiler [4] by Synopsys, etc. Besides, many open source high-level design tools are proposed by academics as well, e.g., LegUP [5], GAUT [6], ROCCC [7], etc. According to the difference of supported source file, these design tools can be categorised into two classes: HLS tools and OpenCL-based high-level design tools.

A. High-Level Synthesis Tools

Most of high-level design tools, which generate RTL code from C-based source code (e.g., C, C++, SystemC), can be categorised into HLS tools, such as Stratus HLS, Vivado HLS, ROCCC, LegUP, etc. HLS has been studied for years in research area and only recently starts to be utilised for real projects. LegUp is one of the best open source high-level synthesis tool being developed by scholars. LegUp framework allows researchers to improve C to Verilog synthesis without building an infrastructure from scratch [8]. However, presently, only few Altera FPGA boards are supported by LegUP compiler and ARM-accelerator hybrid synthesis mode is under development. Vivado HLS, released by Xilinx, is the most popular commercial HLS tool in market. It gives user the possibility to speedup IP creation by enabling C, C++ and System C specifications to be directly targeted into Xilinx FPGAs without the need to manually create RTL [1].

B. OpenCL-based High-Level Design Tools

OpenCL-based high-level design, which employs OpenCL [9] as the programming language, is a relatively new methodology for application design on FPGA platform. OpenCL is an open standard for parallel programming of heterogeneous systems which can consist of CPU, GPU, DSP, FPGA, etc. For a typical SoC-FPGA-based embedded system, the system is usually composed of ARM core, memory and accelerators that generated by programming logics. If we consider the ARM core being the host and accelerators being the devices, the SoC system can be deemed to a heterogeneous system. Therefore, utilising OpenCL as the high-level design language for SoC-FPGA is also promising.

Altera SDK for OpenCL is the only commercial tool that supports OpenCL-based high-level design currently. It provides user a much faster and higher level software development flow for Altera FPGAs by abstracting away the traditional hardware FPGA development workflow [2]. Meanwhile, research that intends to explore open source framework for OpenCL on FPGA is also undergoing. For instance, in [10], an simple OpenCL framework is proposed to evaluate the interconnect implementation on FPGAs.

Recently, research works that utilise or evaluate high-level design tools emerged as well. In [11], a tri-diagonal matrix algorithm is implemented with VHDL, Altera SDK for

2nd International Workshop on *FPGAs for Software Programmers (FSP 2015)*, London, United Kingdom, September 1, 2015.

OpenCL, and Vivado HLS, respectively, and then the generated hardware performance is compared. The possibility of utilising HLS tools in computational finance field are explored in [12]. LegUp, Altera SDK for OpenCL, Bluespec SystemVerilog [13], and Chisel [14] design tools are evaluated for database application accelerations purpose in [15]. All papers conclude that high-level design tools can dramatically shorten the developing time and generate proper hardware. However, all these work are targeting for CPU + FPGA platform. For SoC-FPGA platform which contains ARM + FPGA, no publish research result is available yet.

In order to evaluate development experience of high-level design tools and performance of generated hardware architecture on SoC-FPGA, especially targeting for embedded vision applications, disparity map calculation is selected as the evaluation application; Vivado HLS and Altera SDK for OpenCL, which are the most mature and commercialised tools available, as representative design tools; and Zedboard and SoCKit development board as the corresponding implementation platform, respectively.

The rest of this paper is structured as follows: Section II mainly introduces disparity map calculation application. Thereafter, the design and implementation process of disparity map calculation with these two tools are explained and compared in Section III. Finally, conclusions are drawn in Section IV.

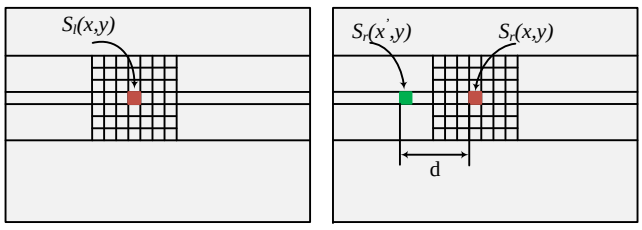
II. Disparity Map Calculation

Disparity map calculation, which also known as depth estimation, is a fundamental but important algorithm widely used in stereo vision systems.

For a stereo vision system, assuming a 3D world point S, its projected points on stereo images being $S_l(x, y)$ and $S_r(x', y)$, as shown in Figure 1. According to the geometry of stereo vision system, the axes of projected points $S_l(x, y)$ and $S_r(x', y)$ are different. To measure this axes difference, disparity value d is introduced and can be computed by

$$d = x - x'. \tag{1}$$

Assuming $S_l(x, y)$ is known, then the process of calculating disparity d is actually to locate the matching point $S_r(x', y)$. In other words, disparity value calculation for each point is a stereo matching process. And the process of computing disparity value for all points between a pair of stereo images is disparity map calculation algorithm.



(a) Arbitrary point in left image (b) Locating corresponding point in right image

Fig. 1: SAD-based stereo matching.

In order to accurately calculate disparity value for each pixel, various stereo matching algorithms, feature-based approaches (e.g., SIFT [16], SBM [11], etc.) and intensity-based approaches (e.g., SAD [17], SSD, etc.) are proposed. Here we don't want to explain in detail or compare different disparity map calculation algorithms since this is out of our topic. For the sake of simplicity, most typical Sum of Absolute Difference (SAD)-based stereo matching algorithm [17] is employed and implemented.

Referring to Figure 1, if we use a $N \times N$ floating window for each image with $S_l(x, y)$ and $S_r(x, y)$ being their respective centers, differences exist between two window blocks. However, if the center points being $S_l(x, y)$ and $S_r(x', y)$, respectively, little differences should exist. To measure these intensity differences, Sum of Absolute Difference (SAD) is introduced and calculated. If we denote $T_d(x, y)$ as the SAD of $S_l(x, y)$ and $S_r(x-d, y)$, $T_d(x, y)$ can be computed by

$$T_d(x,y) = \sum_{(i,j)\in N} |\, S_l(x+i, y+j) - S_r(x-d+i, y+j) \,|\,. \tag{2}$$

Since the matching pixel in right image $S_r(x', y)$ should be on the left side of $S_l(x, y)$, pixel by pixel left shifting of the floating window is executed and the SAD is calculated for each shifting. Assuming the maximum disparity as M, after M times shifting and computing, the minimum SAD $T_(x, y)$ can be obtained by

$$T(x,y) = \min_{d\in M} T_d(x,y), \tag{3}$$

and its corresponding disparity as the true disparity value at point (x, y). Repeat this process for each pixel until disparity map of the whole image is obtained.

III. Implementation on SoC-FPGA

After we are familiar with the SAD-based disparity map calculation algorithm, in this section, disparity map calculation accelerator is designed and implemented with both Vivado HLS and Altera SDK for OpenCL tools. The detailed implementation process is explained as follows.

A. Accelerator Design with Vivado HLS

From the SAD-based disparity map calculation process description in Section II, we can notice that, for each pixel, M times SAD operations should be executed and each SAD operation is composed of $N \times N$ pixels reading from each image, $N \times N$ times absolute difference calculation, and $N \times N - 1$ time absolute difference addition. It is obvious that too much memory accesses and computing logics are required for each pixel processing, which can be difficult for efficient hardware architecture generation on SoC. Therefore, some optimization techniques, especially for FPGA hardware implementation, are performed.

1) Preserving Intermediate Results: Originally, the absolute difference for each pixel-pair will be calculated $N \times N$ times. By preserving the absolute difference results, absolute difference operation for each pixel-pair only need to be executed once, therefore, much less computation resources are required. Meantime, simpler computation logic can simplify the pipeline architecture generated as well. The only price of

preserving intermediate results is that more local memories are required for intermediate data preservation. But comparing to the computation quantity that simplified, the cost is pretty small. In our application, column-SAD is the actual intermediate result being preserved, since it is difficult to explain column-SAD before describing the disparity map calculation process, it will be explained later in this section.

2) Utilising Local Memory: In disparity map calculation application, $N \times N$ pixels from each image should be read in in each pixel processing cycle, which is impossible due to the memory ports limitation. Utilising local memory, especially for image processing applications, which can store a few lines of image pixels, can dramatically improve the pixel accessing performance. The local memory size can be calculated by delicate hardware design. In our case, $N \times N$ floating window for each pixel pair is processed, if one pixel read in and one pixel processing is performed each cycle, N lines pixels storage for each image are required. One pixel from each image is read in per cycle, when the fifth pixel of last row of local memory is read in, disparity map calculation process starts. If local memory is full-filled with pixels, new data will be stored from the first element of local memory again. If we denote the image width as w, the local memory size should be $2N \times w$. Besides, intermediate results can also be stored in local memory.

3) Utilising Shifting Registers: Utilising Shifting registers is an effective approach commonly used in FPGA design with Verilog/VHDL. Vivado HLS also provides the support for hardware generation shifting registers by simply adding directives. Shifting all registers left or right and then reading in new data guarantees that the latest data is always stored at the same location of register array. This can greatly simplify the indexing problem in local memory and is extremely useful for our col-SAD register table. Moreover, efficient/pipelined hardware architecture is easier to be generated with shifting registers utilisation.

4) Other Techniques: Some other techniques are deployed in our application as well, e.g., loop unrolling, pipeline declaration, resource core specification. Since these directives are simple and straightforward, the explanations will be omitted here.

With the techniques mentioned above, the disparity calculation processes as follows. Referring to Figure 2, $N \times w$ size local memory LM and RM are used as N-line-pixels buffers for left and right image, respectively. LR is a N size register array that used to store one column pixels which will be read from LM local memory. RR_i is an M column registers table that each column can preserve one column pixels from RM local memory, with $i = \{0, 1, \ldots, M-1\}$. col-SADs is an $M \times (N+1)$ size shifting registers table which is used to store column-SADs.

For each cycle, the right image buffer RR_i and col-SADs perform one-pixel-left column-shifting. Thereafter, one pixel is read in from each image to local memory LM and RM, respectively. And one column pixels are read in from local memory LM and RM to register array LR and last column of register table RR_i, respectively. Column SADs are calculated between LR and each column of RR_i. The results, according to different disparities, are stored into the last column of col-

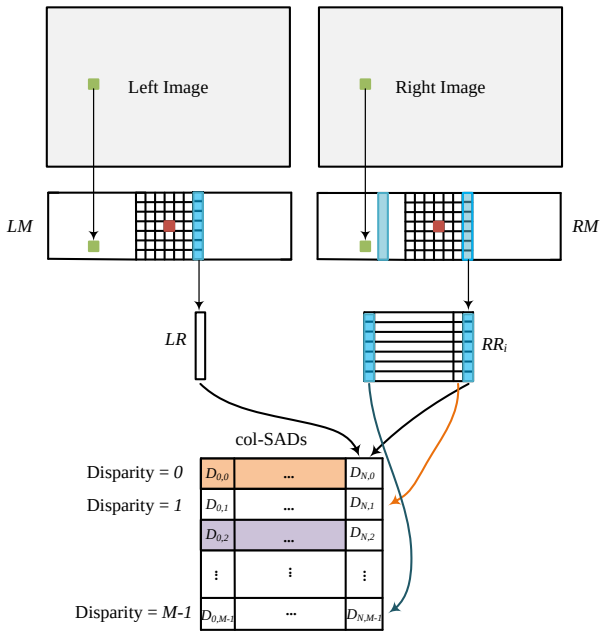


Fig. 2: Disparity map calculation process illustration.

SADs register table. For instance, the col-SAD of LR and RR_0 is stored in the first element of last column of col-SADs $D_{N,0}$; LR and RR_{M-1} col-SAD result is stored in $D_{N,M-1}$; so on and so forth. Meanwhile, the first N columns (0 to $N-1$) of col-SADs perform row sum, which is to get the SAD for each disparity. Finally the smallest sum corresponding disparity is selected as the real disparity at this point.

This approach reduces the memory access by using local memory and reduces computation by preserving col-SADs. Although, for every new line, all data in shifting registers need to be reloaded or recalculated. But since the computing process is pretty suitable for pipeline processing, efficient hardware architecture can be generated.

In this paper, 640×480 stereo images are used as input and Zedboard is deployed as the targeting SoC board. After we connect HLS generated accelerator with ARM core in Vivado, generate bitstream and on board testing, the results are finally obtained and listed in Table I. As Table I shows, different window size and max disparity settings are set up and the resources usages on Zedboard are listed as well. The reason we select these setups is in order to make the comparison with Altera SDK for OpenCL generated hardware. Because many shifting registers and computing resources are utilised in the algorithm, therefore, a lot of logics are used on Zedboard. As for the memory usage, only two N-line-pixels are stored in local memory, which is relatively little portion on Zedboard. The last column of Table I lists the running FPS of disparity map calculation accelerator on Zedboard. It is obvious that FPS 228 doesn't really change with different setups. The reasonable explanation for that is deep pipelined hardware is generated for each setup, which means one pixel processing each clock cycle is truly executed. Therefore, for the same resolution rate stereo images, similar running FPS should be obtained on board.

Setup		Resource Usage			FPS
Window Size	Max Disparity	Logic	Mem	DSP	
7×7	80	60.8%	5%	0%	228
9×9	64	59.8%	6%	0%	230
9×9	78	83.3%	6%	0%	228
9×9	80	84.6%	6%	0%	228

TABLE I: Results of disparity map calculation with Vivado HLS generated accelerator.

Setup		Resource Usage			FPS
Window Size	Max Disparity	Logic	Mem	DSP	
7×7	80	75%	23%	0%	242
9×9	64	82%	29%	0%	208
9×9	70	89%	29%	0%	198
9×9	78	98%	29%	0%	193

TABLE II: Results of disparity map calculation with Altera SDK for OpenCL generated accelerator.

B. Accelerator Design with Altera SDK for OpenCL

Unlike C++ as our application source file type in Vivado HLS, Altera SDK for OpenCL utilises OpenCL as its unified programming language for both hardware and software, therefore, some OpenCL framework setup operations, such as platform setup, memory setup, etc., should be added into the original source file first. However, before touching the kernel code with OpenCL, one thing we should determine is which kind of kernel structure shall be deployed: single work-item kernel or NDRange kernel.

Altera SDK for OpenCL supports two different types of kernel, NDRange kernel and single work-item kernel. Single work-item kernel can also be seen as a $(1, 1, 1)$ NDRange kernel. But for Altera SDK for OpenCL, different directives and compiler behaviours are supported for different type of kernel. Therefore, proper kernel type should be deployed according to application characteristics. For the application with lots of data dependency or many loops, single work-item kernel can generate efficient pipelined hardware architecture and boost up the performance. On the contrary, if little data dependency or loops exist in the application, NDRange kernel can execute multiple Processing Elements (PEs) in parallel and improve the processing efficiency [18]. In our case, according to the calculation process in Section II, nested loops and data dependency exist. Therefore, single work-item kernel should be a better choice for our application. As a matter of fact, for single work-item kernel, the kernel code is similar to the original C code, therefore, only some specifications/directives are required.

Since the application is already well optimised with some techniques and directives in Section III-A, similar techniques are used in Altera design tool version as well. However, unlike Vivado HLS, Altera SDK for OpenCL doesn't provide any assistant window to help with the directive addition, therefore, all the directives must be added manually according to the user guide document [18] [19]. Thereafter, setting up same parameters (window size and max disparity) as in Vivado HLS, the whole system (hardware and software) is generated for SoCKit Development board. Unlike Vivado HLS, no further design or processing is required for embedded system generation. After on board testing with generated hardware architecture, the resource usage and performance results are obtained and listed in Table II.

From Table II, we can state that, efficient hardware accelerator can be generated with Altera SDK for OpenCL as well. But due to the resource limitation on SoCKit, which is actually similar to Zedboard, with 9×9 window size setting-up, only 78 maximum disparity can be achieved. Moreover, Table II also shows that, comparing to Vivado HLS generated hardware, around 10% more programming logics are use. As for memory usage, in OpenCL programming model, the whole stereo images are stored in the allocated global memory, therefore, much more memory is and should be utilised.

C. Comparison

After we design and implement disparity map calculation application with Vivado HLS and Altera SDK for OpenCL, respectively, we can conclude that, both tools can generate efficient hardware architecture with much shorter developing time. But the tools themselves and development experiences with them are actually quite different. Vivado HLS is a tool that used to speedup hardware accelerator (IP) design and creation on FPGA. The generated file is IP core, which can be connected to other IP or ARM in Vivado system design. Therefore, some work should be done in Vivado as well and hardware knowledge is necessary for the user. Besides, more directives, such as memory type, interface, loop control, etc., are supported in Vivado HLS. By contrast, Altera SDK for OpenCL is a tool provided for software programmers, which means little hardware knowledge is required for user and much easier workflow for the whole process. However, comparing to Vivado HLS, less directives are supported and more difficult to debug applications. Of course, OpenCL knowledge is required for the programmer.

IV. Conclusion

This paper explores high-level design approaches on SoC-FPGA platform with disparity map calculation as the targeting application. Vivado High-Level Synthesis (HLS) and Altera SDK for OpenCL, two representative and most mature commercial tools, are selected as the design tools. Zedboard and SoCKit development board are the corresponding implementation SoC-FPGA, respectively. Hardware accelerators of disparity map calculation are designed with Vivado HLS and Altera SDK for OpenCL, respectively, and comparisons are made for the implementation process and generated hardware performance. From the comparison, we conclude that, both design tools can generate efficient hardware for disparity map calculation application with much less developing time. For 640×480 stereo images, window size being 9×9, and max disparity being 80 (78 in Altera SDK for OpenCL), 228 and 193 fps disparity map calculation can be achieved, respectively. However, for same algorithm and setups, more resources on FPGA are used for the accelerator implemented by Altera SDK for OpenCL. Besides, more hardware knowledge is required for Vivado HLS user and Altera SDK for OpenCL is more suitable for software programmers.

REFERENCES

[1] "Xilinx High-Level Synthesis," http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[2] "Altera SDK for OpenCL," https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html.

[3] "Stratus High-Level Synthesis," http://www.cadence.com/products/sd/stratus/pages/default.aspx.

[4] "Synphony C Compiler," https://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/SynphonyC-Compiler.aspx.

[5] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "LegUp: High-level Synthesis for FPGA-based Processor/Accelerator Systems," in *Proc. of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, New York, NY, USA, Feb. 2011, pp. 33–36.

[6] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin, "GAUT: A High-Level Synthesis Tool for DSP Applications," in *High-Level Synthesis*, P. Coussy and A. Morawiec, Eds. Springer Netherlands, 2008, pp. 147–169.

[7] "RoCCC 2.0," https://www.http://roccc.cs.ucr.edu.

[8] "LegUP," http://legup.eecg.utoronto.ca.

[9] A. Munshi, "The OpenCL Specification." Khronos OpenCL Working Group, 2009.

[10] V. Mirian and P. Chow, "Using an OpenCL framework to evaluate interconnect implementations on FPGAs," in *Proc. of 2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2014, pp. 1–4.

[11] X. Fan, X. Wang, and Y. Xiao, "A Shape-Based Stereo Matching Algorithm for Binocular Vision," in *Proc. of 2014 International Conference on Security, Pattern Analysis, and Cybernetics (SPAC)*, Wuhan, China, Oct. 2014, pp. 70–74.

[12] Q. Gautier, A. Shearer, J. Matai, D. Richmond, P. Meng, and R. Kastner, "Real-time 3D Reconstruction for FPGAs: A Case Study for Evaluating the Performance, Area, and Programmability Trade-offs of the Altera OpenCL SDK," in *Proc. of 2014 International Conference on Field-Programmable Technology (FPT)*, Shanghai, China, Dec. 2014, pp. 326–329.

[13] "Bluespec, Inc," http://www.bluespec.com/.

[14] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: Constructing hardware in a Scala embedded language," in *Proc. of 2012 49th Design Automation Conference (DAC)*, June 2012, pp. 1212–1221.

[15] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal, and M. Lujan, "An Empirical Evaluation of High-Level Synthesis Languages and Tools for Database Acceleration," in *Proc. of 24th International Conference on Field Programmable Logic and Applications (FPL)*, Munich, Germany, Sept. 2014, pp. 1–8.

[16] D. Lowe, "Object Recognition From Local Scale-Invariant Features," in *Proc. of the Seventh IEEE International Conference on Computer Vision*, vol. 2, Kerkyra, Greece, Sept. 1999, pp. 1150–1157.

[17] S. Vassiliadis, E. Hakkennes, J. Wong, and G. Pechanek, "The Sum-Absolute-Difference Motion Estimation Accelerator," in *Proc. of 24th Euromicro Conference*, vol. 2, Vasteras, Sweden, Aug. 1998, pp. 559–566.

[18] "Altera SDK for OpenCL Best Practices Guide," May 2015.

[19] "Altera SDK for OpenCL Programming Guide," May 2015.

RIPL: An Efficient Image Processing DSL for FPGAs

Robert Stewart & Greg Michaelson
Mathematical & Computer Sciences
Heriot-Watt University
Edinburgh, UK
{R.Stewart,G.Michaelson}@hw.ac.uk

Deepayan Bhowmik & Andrew Wallace
Engineering & Physical Sciences
Heriot-Watt University
Edinburgh, UK
{D.Bhowmik,A.M.Wallace}@hw.ac.uk

***Abstract*—Field programmable gate arrays (FPGAs) can accelerate image processing by exploiting fine-grained parallelism opportunities in image operations. FPGA language designs are often subsets or extensions of existing languages, though these typically lack suitable hardware computation models so compiling them to FPGAs leads to inefficient designs. Moreover, these languages lack image processing domain specificity. Our solution is RIPL[1], an image processing domain specific language (DSL) for FPGAs. It has algorithmic skeletons to express image processing, and these are exploited to generate deep pipelines of highly concurrent and memory-efficient image processing components.**

I. Introduction

FPGAs can be configured directly with hardware description languages, though these require hardware expertise and come with the cost of long debugging stages to remove design errors. Alternatives include high level synthesis tools to compile existing imperative languages, and dataflow languages that abstract the highly concurrent nature of FPGA hardware. However, the absence of suitable hardware to support the imperative model make compiling them to FPGAs very inefficient, and dataflow languages burden programmers with wiring together computations explicitly.

RIPL abstracts dynamic dataflow process networks (DPNs) by hiding actors and wires, and inherits DPN hardware abstractions of clocks, signals, registers and memory. The RIPL programmer uses a collection of built in image processing skeletons, and the compiler automatically extracts parallelism from the program, to generate deeply pipelined and memory efficient FPGA designs.

II. Design

A. Requirements & Constraints

Higher order computer vision algorithms are composed of lower level image operations. Prototypical image processing operations can be classified in terms of the locality of their data access requirements: pixel to pixel functions on *points*, neighbourhood pixels to pixel functions on *regions*, and *global* operations on entire images.

The memory constraints of FPGAs mean that many CPU & GPU methods for parallel image processing cannot be adopted for FPGA image processing implementations. Software techniques often store arrays whose sizes matches complete images, and apply data-parallel kernels in a vectorised single instruction multiple data (SIMD) or coarse grained single instruction multiple threads (SIMT) fashion. These image processing models are prohibitive for FPGA implementations, because on-chip memory is a very scarce resource, and the global shared memory model is not suitable for the inherently fine grained concurrent nature of FPGAs. Modern CPUs have access to around 2MB cache and 64GB of RAM, which are treated as a large shared memory block. In contrast for example, a modern Virtex 7 FPGA has a total of just 8.5MB of available on-chip block RAM (BRAM) memory.

B. RIPL Overview

RIPL is a functional language with single assignment semantics. It comprises domain specific image processing types, functions and algorithmic skeletons [1]. RIPL's algorithmic skeletons are reusable parameterised descriptions of task-specific image processing architectures and are exploited to generate pipelines of image operations. The skeletons process pixel vectors in rows, columns and regions with computation kernels, which are lightweight functions that traverse over images.

An illustration of pipelined skeleton composition is in Figure 1. The RIPL skeletons API is shown in Figure 2, using standard notation for function type signatures, *e.g.* `mapRow` takes as arguments: an $M \times N$ image, a function from a vector of A pixels to a vector of A pixels, and returns an $M \times N$ image. The `map` skeletons are element or column/row wise mappings from pixels to pixels. The `zipWith` skeleton takes two images and merges them into a single stream with some user defined merging function. The `combine` skeleton takes entire rows or columns from two images and merges them into a single stream with built-in RIPL operator, such as `append`. The `convolve` skeleton is parameterised by a window dimension and computes pixel values from a neighbourhood of pixels. The `fold` skeleton is parameterised by an initial value and applies global operations over an image and returns a scalar value or a vector.

RIPL uses index types to impose the constraint that all skeletons operate on images with bounded shapes known at compile time. For example, an inferred indexed data type $Im_{(50,40)}$ is an image of width 50 and height 40, and $[P]_8$ is a vector of 8 pixels. This allows the RIPL compiler to generate actors with static arrays from these

[1]Rathlin Image Processing Language

2nd International Workshop on *FPGAs for Software Programmers (FSP 2015)*, London, United Kingdom, September 1, 2015.

indexed data structures, enabling HDL synthesis tools to make optimal memory implementation choices about static structures, *i.e.* look-up tables (LUTs) for small arrays or with combined BRAM blocks for larger arrays.

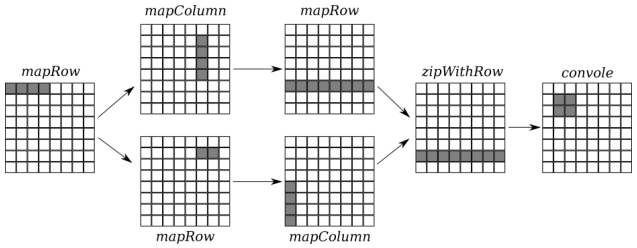


Fig. 1. Pipelining skeleton compositions

$$
\begin{aligned}
mapRow &: Im_{(M,N)} \to ([P]_A \to [P]_A) \to Im_{(M,N)} \\
mapCol &: Im_{(M,N)} \to ([P]_A \to [P]_A) \to Im_{(M,N)} \\
concatMapRow &: Im_{(M,N)} \to ([P]_A \to [P]_B) \to Im_{(B/A*M,N)} \\
concatMapCol &: Im_{(M,N)} \to ([P]_A \to [P]_B) \to Im_{(M,B/A*N)} \\
zipWithRow &: Im_{(M,N)} \to Im_{(M,N)} \to (P \to P \to P) \to Im_{(M,N)} \\
zipWithCol &: Im_{(M,N)} \to Im_{(M,N)} \to (P \to P \to P) \to Im_{(M,N)} \\
combineRow &: Im_{(M,N)} \to Im_{(M,N)} \\
&\to ([P]_A \to [P]_A \to [P]_B) \to Im_{(B/A*M,N)} \\
combineCol &: Im_{(M,N)} \to Im_{(M,N)} \\
&\to ([P]_A \to [P]_A \to [P]_B) \to Im_{(M,B/A*N)} \\
convolve &: Im_{(M,N)} \to (a,b) : (Int, Int) \\
&\to ([P]_{a*b} \to P) \to Im_{(M,N)} \\
foldVector &: Im_{(M,N)} \to Int \to s : Int \\
&\to (P \to [Int] \to [Int]) \to [Int]_s \\
foldScalar &: Im_{(M,N)} \to Int \to (P \to Int \to Int) \to Int
\end{aligned}
$$

Fig. 2. RIPL skeletons API

III. Implementation

A. RIPL to Dataflow

The RIPL compiler uses the dynamic dataflow process network (DPN) model as an intermediate representation between the DSL and FPGA implementations. To address FPGA memory limitations, the compiler eliminates intermediate image arrays by feeding rows and columns through concurrent phases of a pipeline. The kernel at each phase is provided only the pixel values they need to execute. Costly intermediate arrays are therefore avoided for local and regional data access patterns with RIPL skeletons.

Thanks to RIPL's single assignment semantics, implicit data dependencies in skeleton compositions are exploited to generate deeply pipelined graphs from RIPL programs. The vertices (actors) represent image operations and the edges (wires) represent dataflow between composed operations. Transposition actors are added whenever a row wise skeleton is composed with a column wise skeleton, and vice versa. Informally, the mapping from skeletons to graphs is as follows. A skeleton instance maps to one actor. The arity of a skeleton maps to the number of input ports the corresponding actor has. The number of output ports of an actor is dictated by the number of other skeletons that use the output image of the skeleton. Implicit dataflow in the composition of skeletons is lifted to explicit wires between actors. The user defined function to a skeleton becomes a fireable rule inside the actor. The graph is mapped onto FPGAs to exploit two kinds of pipelined parallelism: 1) to feed the rows/columns of an image through different pipeline stages, and 2) to feed multiple video frames into the FPGA fabric concurrently.

B. Dataflow to FPGAs

The generated dataflow graph is expressed with the CAL actor language [2]. An existing CAL to Verilog compiler [3] is used to add an interface protocol for actor interconnects and an explicit clock to all actor components, then lowers the graph to an FPGA abstraction adding signals, registers, FIFOs and shared memories. Generic memories are used for arithmetic operations, registers and actor interconnects, allowing HDL synthesisers to choose from LUT or BRAM instantiations, depending on holistic memory requirements and on the FIFO depths needed to support implicit dataflow dependencies in RIPL programs.

IV. Discussion & Conclusion

In this abstract we present RIPL, a high level image processing DSL for FPGAs. It has high level image processing skeletons familiar to software programmers, which are exploited to generate deep pipelines of memory-efficient image processing operations. RIPLs underlying dynamic dataflow model supports different image data access patterns using skeletons. The aim of RIPL is to maximise clock frequency to increase throughput, and to minimise BRAM use to fit complex algorithms onto FPGAs. RIPL has been used to implement image watermarking and multi-dimensional subband decomposition algorithms. We believe that RIPLs underlying dynamic DPN semantics provides greater levels of expressivity compared to other image processing FPGA languages. Ongoing work includes evaluating the expressivity of RIPL with a comprehensive collection of case studies. We plan on integrating RIPL with a performance guided dataflow transformations framework we are developing [4].

Acknowledgements

We acknowledge the support of the Engineering and Physical Research Council, grant references EP/K009931/1 (Programmable embedded platforms for remote and compute intensive image processing applications).

References

[1] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1991.

[2] J. Eker and J. W. Janneck, "CAL Language Report Specification of the CAL Actor Language," EECS Department, University of California, Berkeley, Tech. Rep. UCB/ERL M03/48, 2003.

[3] E. Bezati, "High-Level Synthesis of Dataflow Programs for Heterogeneous Platforms: Design Flow Tools and Design Space Exploration," Ph.D. dissertation, School of Engineering, Ecole Polytechnique Federale de Lausanne, Switzerland, April 2015.

[4] R. Stewart, D. Bhowmik, G. Michaelson, and A. Wallace, "Profile Guided Dataflow Transformation for FPGAs & CPUs," *Dataflow special issue in the Journal of Signal Processing Systems*, August 2015.

GCC-Plugin for Automated Accelerator Generation and Integration on Hybrid FPGA-SoCs

Markus Vogt, Gerald Hempel, Jeronimo Castrillon
Technische Universität Dresden
Faculty of Computer Science
Dresden, Germany
Email: {forename.surname}@tu-dresden.de

Christian Hochberger
Computer Systems Group
Technische Universität Darmstadt
Darmstadt, Germany
Email: hochberger@rs.tu-darmstadt.de

***Abstract*—In recent years, architectures combining a reconfigurable fabric and a general purpose processor on a single chip became increasingly popular. Such hybrid architectures allow extending embedded software with application specific hardware accelerators to improve performance and/or energy efficiency. Aiding system designers and programmers at handling the complexity of the required process of hardware/software (HW/SW) partitioning is an important issue. Current methods are often restricted, either to bare-metal systems, to subsets of mainstream programming languages, or require special coding guidelines, e.g., via annotations. These restrictions still represent a high entry barrier for the wider community of programmers that new hybrid architectures are intended for. In this paper we revisit HW/SW partitioning and present a seamless programming flow for unrestricted, legacy C code. It consists of a retargetable GCC plugin that automatically identifies code sections for hardware acceleration and generates code accordingly. The proposed workflow was evaluated on the Xilinx Zynq platform using unmodified code from an embedded benchmark suite.**

I. Introduction

Today, embedded hybrid platforms combining field programmable gate arrays (FPGA) and high performance RISC processing cores give the user the freedom to implement specialized peripherals in the FPGA fabric while still relying on the execution power of the RISC processor(s). The Xilinx Zynq system on chip (SoC) family and the Altera Cyclone/Arria V SoC are prominent examples for this approach.

Such devices pave the path for the integration of arbitrary hardware accelerators in complex applications, however, most software developers are not familiar with hardware description languages (HDL). Thus, they are unable to develop application specific accelerators on their own. This problem has been addressed in the past by many researchers. Yet, the proposed solutions are not satisfactory. The user still has to write his own HDL code, has to take care of the HW/SW partitioning (often by annotating the existing code) and has to create the required SW/HW interfaces.

Our approach aims to notably lower the entry barrier for software developers to hardware-accelerated program execution. This particularly means using plain unannotated C, which is a popular and established language, as input. In this way, we bring hardware acceleration to a broader range of general applications. We envision a transparent workflow ideally not demanding any HDL skills or knowledge about the underlying hardware platform from the developer, providing seamless integration with the software environment.

The contributions of this paper are:

- Automated HW/SW partitioning using a GCC-plugin that extracts accelerators from C code and generates synthesizable HDL code.
- Automated and platform agnostic code patching enables seamless integration with software environment. Accelerator invocation remains completely transparent with optional fall-back to software execution.
- Support for legacy application code without annotations.

The rest of this paper is organized as follows. Section II presents the related work. In Section III we introduce the target hardware platforms of our proposed workflow. Section IV describes our workflow, the compiler plugin and its integration into GCC. Sections V and VI present the evaluation of our approach and discuss results. The remaining Sections draw conclusions and point out future work.

II. Related Work

Since the emergence of FPGAs, many efforts have been made to exploit the performance gain offered by reconfigurable logic with customized hardware accelerators. This especially holds true for hybrid FPGA architectures tightly coupling a general purpose processor with reconfigurable logic.

The most obvious, flexible but also the most challenging way is to write accelerators by hand using an HDL and manually perform all required integration with the software environment. An example is shown in [1]. Designing accelerator-based systems that way, requires strong skills in HDL as well as deep knowledge of the underlying hardware platform. The development process usually is time consuming and error-prone. Hence, the ability to implement such systems is left to the relatively small community of FPGA developers.

A number of approaches have been presented that reduce or even completely eliminate the necessity of writing HDL. The goal is to generate synthesizable code for accelerators from a more abstract problem description. LegUp [2] is an open source high-level synthesis (HLS) tool for FPGA based hybrid systems. The HW/SW partitioning is determined by profiling the C program on a self-profiling processor and altering the software binary afterwards in order to run it on

2nd International Workshop on *FPGAs for Software Programmers (FSP 2015)*, London, United Kingdom, September 1, 2015.

the hybrid system. In [3] the authors present basic support for ARM-FPGA hybrid SoCs. In [4] the authors present Nymble, a system based on the techniques introduced by COMRADE [5]. It allows a much larger scope for accelerators by supporting a mechanism for back-delegation of unsuitable code sections into software. For HW/SW partitioning, Nymble requires additional code annotations using pragmas.

Nymble as well as LegUp use Low Level Virtual Machine (LLVM) as compiler framework. As shown in [6], the Gnu compiler collection (GCC) has been used for HLS workflows as well. The authors show a customized GCC compiler for generation of hardware accelerators for a bare-metal soft-core processor. Our work extends C-to-HDL transformations for better integration in more complex systems.

The Delft Workbench [7] is a toolset providing semi-automatic HW/SW partitioning as well as HLS for FPGA. The targeted Molen machine architecture can be regarded as hybrid FPGA-processor architecture. The candidate kernels for hardware acceleration are determined by profiling but must be extracted manually.

Xilinx provides Vivado [8], [9], one of the most popular commercial HLS tools. It supports translating C, SystemC or C++ code directly into hardware. Vivado aims at mapping the whole application to hardware, which requires manual HW/SW partitioning by the user. Similar to Vivado, other HLS tools like ROCCC [10] or CATAPULT [11] provide sophisticated hardware synthesis for hardware-only solutions, with no support for a hybrid HW/SW translation. In [12] the authors present a framework that matches portions of C code (algorithmic skeletons) exposing specific memory access patterns against a library of known accelerator templates. In [13] authors particularly address the integration of accelerators with the software domain. They present a linker that creates an executable by transparently linking functions implemented in software objects and/or hardware accelerators. With the runtime environment provided, programs can be executed on a Zynq platform running embedded Linux.

Most of the approaches mentioned so far address a certain task related to accelerator generation or integration, but the user still has to perform manual work. This requires, even though to a lesser extent, knowledge of HDL and the underlying hardware platform. In contrast, the work in [14] raises the level of abstraction to completely hide the HW/SW boundary from the software developer. The work in [14] applies the principle of binary acceleration, which means identifying sequences of processor instructions worthy of acceleration at runtime and migrating them to specialized execution units. However, live application analysis and accelerator synthesis typically require a reasonable amount of computational resources, pushing todays embedded runtime environments towards their limits.

A promising solution to the issues left open by the approaches mentioned above is to rethink the entire design flow. This has been done by the Liquid Metal project [15]. The authors developed the Lime language [16], enabling programmers to describe a system in a hardware-friendly but still object-oriented manner. Lime programs can be compiled either into pure software binaries or into software and a set of hardware accelerators. All interfacing is done automatically by the runtime environment. Introducing a well tailored language circumvents limitations that arise from using existing languages. However, adopting a new language is a high entry barrier for most programmers and existing software must be ported to benefit from hardware acceleration.

III. Platform

The work presented in this paper especially addresses recent hybrid platforms combining embedded processors with a reconfigurable fabric. In this section we briefly describe one such system, namely, the ZedBoard evaluation kit containing a Xilinx Zynq-7000 [17] device.

The programmable logic (PL) in the Zynq-7000 device is a full Artix-7 FPGA fabric, while the processing system (PS) is a complete ARM subsystem featuring a Cortex-A9 dual core processor and a comprehensive set of peripherals. The PS provides four 32-bit general purpose (GP) AXI interfaces, which allow connecting peripherals from the PL as well as four full-duplex 64-bit high performance (HP) interfaces for connecting AXI masters residing in the PL. The Zynq architecture provides one special high performance interface connected to the Accelerator Coherency Port (ACP). The ACP is internally connected to the ARM Snoop Control Unit and can be used for cache coherent accesses to the ARM subsystem.

It should be noted, that the specific handling of these different AXI interfaces depends on the hardware residing in the PL which presumes a profound understanding of the hardware accelerator.

IV. Workflow

The workflow for transparent HW/SW partitioning and compilation is composed of four steps as shown in Figure 1. (1) *loop data collection* performs a whole-program analysis collecting information about all loops across all compilation units. (2) *loop analysis* uses that information to select loops for potential HW acceleration, using a cost model of the target platform. (3) *hardware generation* performs an HLS of the loops selected by the previous step and (4) *application modification* adapts the original software code to integrate the accelerators and finally generates the application binaries.

Before discussing the steps in detail in Sections IV-B through IV-E, we briefly describe the compiler framework that was used for the workflow.

A. Compiler Framework and Integration

The workflow in Figure 1 was implemented as two plugins for the GCC C compiler. GCC is one of the most widely used compilers for software development for embedded systems. Using such a mature and widely-used compiler framework helps to provide a full transparent workflow for software programmers.

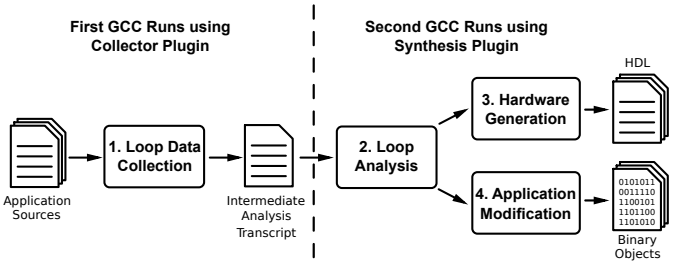


Figure 1. Abstract workflow for automatic accelerator generation

```
1  int fun3(int a, int b);
2  int fun1(int a, int b) {
3      int c;
4      for (int i=0; i<10; i++)
5          c += fun2(b + a, a - b);
6      return c;
7  }
8  int fun2(int a, int b) {
9      for (int i=0; i<30; i++) {
10         a += fun3(a, b);
11         b -= a;
12     }
13     return a+b;
14 }
```

Listing 1. Source code of **unit1.c**

```
1  int fun3(int a, int b) {
2      for (int i=0; i<100; i++) {
3          a += b;
4          if (a > 200 )
5              break;
6          b--;
7      }
8      return a;
9  }
```

Listing 2. Source code of **unit2.c**

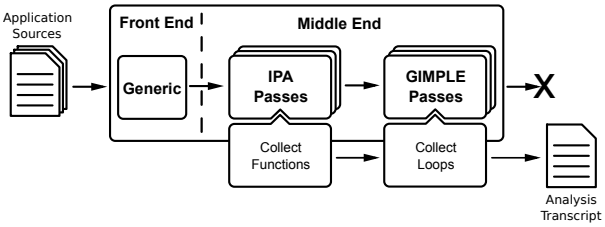


Figure 2. First GCC run invoking the *collector plugin*

GCC follows the traditional compiler structure divided into front-end, middle-end and back-end. Our analysis and transformations are performed in the middle-end on GIMPLE, GCCs internal intermediate representation. GIMPLE basically is a control flow graph (CFG) organized in basic blocks (BB) each containing statements in static single assignment (SSA) form. GIMPLE is further transformed into GCCs internal register transfer language (RTL) which finally is used by the compiler back-end to generate target specific machine code. All internal processing in GCC is controlled by its pass manager, while a pass refers to a certain transformation applied to the internal representation of the current compilation unit. In order to implement the steps depicted in Figure 1, custom passes are inserted using the pass manager.

To reason about the benefits of implementing a certain accelerator, one requires a global view of the application. However, GCC processes each file as separate compilation unit, which hinders whole program analysis. To overcome this drawback, GCC provides a link time optimization (LTO) framework which enables assorted optimizations during link time by storing the GIMPLE representation of each translation unit in the associated object file. Unfortunately, LTO only provides limited hooks for custom passes. We enable whole program analysis without using LTO by running the compilation flow twice (left and right part of Figure 1). The first run collects all information providing the second run with an overall view of the whole application. This global view is required in order to find accelerator candidates. The two consecutive compiler runs are wrapped by GNU Make to remain transparent to the user.

Since version 4.5.0 GCC provides a plugin interface for custom optimization passes, which are invoked by the pass manager using callback functions. The passes described in the following sections are implemented as two plugins for GCC 4.8.3, the *collector plugin* and the *synthesis plugin*. As the plugins work with an existing compiler binary, building a cross compiler for the target architecture is not required. Nevertheless, the plugins themselves must be built for a specific target architecture and hardware interface. Currently, we support the ARM architecture with AXI bus interface and an additional FPGA-based soft-core processor [18] using its proprietary bus interface.

B. Loop Data Collection

The first GCC run invokes the *collector plugin* (step 1 in Figure 1), which implements the two custom passes *collect functions* (CF) and *collect loops* (CL) as shown in Figure 2. The CF pass is executed after all the inter-procedural passes (IPAs) have run. At this point, the compiler knows all functions declared and called in the translation unit. This information is preserved for later use. The CL pass runs after the GIMPLE loop optimizer passes, when all loops in the translation unit have been processed by the compiler. We now collect the compilers internal profiling data for each loop, which includes the local iteration count and a list of called functions and accessed memory locations. The data gathered by the passes CF and CL is accumulated in a single analysis transcript file.

Listing 3 shows a simplified version of the analysis transcript file after compiling the source files shown in Listings 1 and 2. The property `well_nested` indicates whether a loop or loop nest is synthesizable at all. Since loop 1 and 2 both include function calls, `fun2` and `fun3` respectively, only loop 3 will further be considered for accelerator generation.

C. Loop Analysis

The second GCC run invokes the *synthesis plugin* (steps 2, 3 and 4 in Figure 1) which implements the custom pass

```
unit1.c
 function=fun2
  loop1
   count=29
   call=1
   well_nested=0
   -fun3
 function=fun1
  loop2
   count=9
   call=1
   well_nested=0
   -fun2

unit2.c
 function=fun3
  loop3
   count=99
   call=0
   well_nested=1
```

Listing 3. Analysis transcript file after compiling **unit1.c** and **unit2.c**

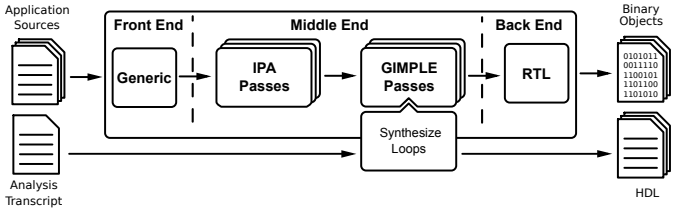


Figure 4. Second GCC run invoking the *synthesis plugin*

synthesize loops (SL) as shown in Figure 4. To ensure consistency of GCCs internal GIMPLE representation, the passes CL (first run) and SL (second run) must be invoked at the same processing stage within each run. In step 2 the SL pass reads the transcript file written by the collector plugin and constructs a call graph for the whole application. Based on this graph the total iteration count for each loop is estimated. Now, by default, all loops are ordered by their total iteration count in order to select the first n loops or loop nests as synthesis candidates. The value for n is a runtime parameter for the compiler plugin and the sorting function is customizable to consider other loop properties, e.g. instruction count. This enables the implementation of an arbitrary cost model to sort the loops.

D. Hardware Generation

In step three of our workflow we generate an HDL implementation for all loops selected by the previous step. This is accomplished by translating the GIMPLE CFG of each loop or loop nest into a finite state machine (FSM). If this step discovers GIMPLE statements or operands which cannot be handled, a compiler warning is generated and the loop candidate is rejected.

During the generation of the FSM a number of optimization techniques are applied. Namely, speculative execution of conditional branches in parallel, list or modulo scheduling, and chaining of consecutive arithmetic operations. We do not explicitly address resource sharing, since FPGA vendor tools achieve better results for that purpose [19].

We are able to estimate the number of clock cycles for worst and best case execution as we know the clock cycle overhead of the accelerator invocation, the clock frequency ratio to the host processor and the shortest and longest path of our FSM. Furthermore, we define an architecture-specific penalty for memory accesses. Along with these heuristic, we are able to estimate the speedup of the accelerator in question. If the results do not meet the constraints specified as compilation parameters, the accelerator is rejected.

The final HDL implementation of the accelerator consists of a loop specific and a target specific part. The former implements a combination of FSM and datapath with a generic register and memory interface. The latter adopts this interface to a target specific host processor interface, e.g. a certain peripheral bus architecture. For example on Zynq, the accelerator is integrated as AXI peripheral module into the system.

E. Application Modification

The final step of our workflow modifies the original code in order to call the synthesized accelerators from the application. We use an abstract calling scheme from the applications point of view. This decouples the code patching from the actual communication protocol. Therefore, each accelerator invocation is wrapped by a generated function. Its implementation is emitted as C code and provides input and output arguments for data transfer between application and accelerator. In our implementation, this function determines the base address of the called accelerator, writes input values to registers, starts the accelerator and reads back output values on return.

The call to that wrapper function is placed preceding the BB of the original loop header, as depicted in Figure 5. It shows the original and modified GIMPLE graphs of the loop in `fun3()` (Listing 2). The inserted variables `tmp.9` and `tmp.16` provide the return values from hardware. They correspond to the original loop exit variables `a.6` and `a.7` respectively. To properly retain control flow in case of multiple loop exits, the accelerator always returns `bb_idx`, which denotes the basic block of the original loop the exit condition occurred in. This value is evaluated by inserted conditional branches directing control flow to the corresponding BB after the original software loop. This bypass is inserted between the wrapper function call and the original loop header. Preserving the original loop enables fall-through to software execution. Resource allocation and sharing techniques can be then applied, since the application is still functional in case no accelerator is currently available.

The presented calling scheme requires inserting a few GIMPLE instructions only while providing the whole flexibility of C for implementing the actual hardware access. This further manifests when targeting platforms running a full-blown OS, such as Linux on the Zynq platform. Such systems require invocation of device drivers for accessing the underlying hardware, which could be rather complex. Furthermore, the wrapper function could be modified or extended by arbitrary user code with little effort. This especially enables debugging of the accelerator call using additional code or even breakpoints.

V. Evaluation

The evaluation presented in this section pursues three different goals, namely, (i) test the prototype of our seamless programming flow on a sample application, (ii) show the generality of our approach by applying it to arbitrary unmodified

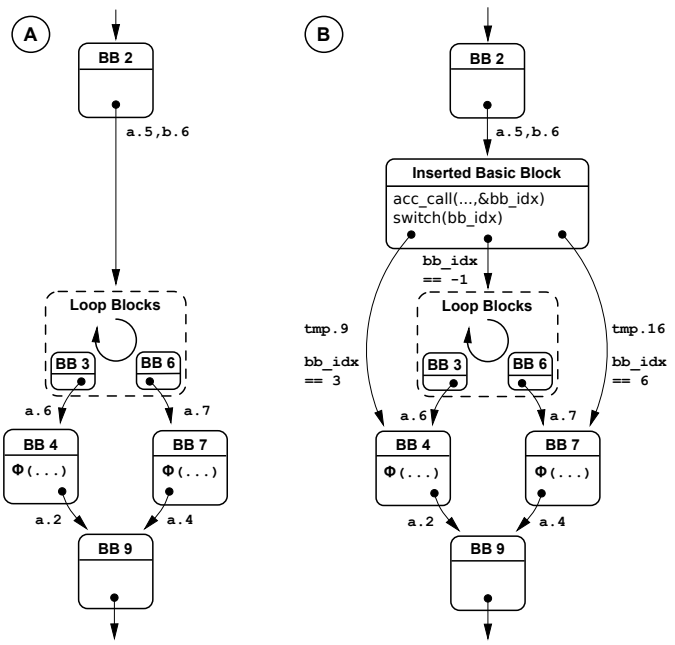


Figure 5. (A) Original GIMPLE Graph; (B) Modified GIMPLE graph with accelerator call

code, and (iii) analyze the challenges posed by today's hybrid platforms for automated accelerator generation (Section VI).

A. Line of Sight

The operability of our approach was demonstrated with a 2D line-of-sight algorithm which determines whether a line intersects a square by iteratively checking each point on the line. Our GCC workflow transforms the main loop of this algorithm into an FSM with 10 states. The generated interface used 11 input registers and one output register in order to exchange data with the accelerator.

Our prototype was implemented on a ZedBoard running Arch Linux. The ARM processor was clocked at 666 MHz, while the accelerator operated at 333 MHz. Both components were connected via a GP AXI port. Using an HP port would not have improved performance, since we have not yet implemented accelerator controlled memory access. We tested our implementation using random input data to eliminate run time dependencies on the data. To prove correctness, we compared the results calculated in software with those calculated by the accelerator hardware. Table I shows the execution times of the accelerator and software-only version. The overhead of an accelerator call is $\approx 2.4\,\mu$s. This value is composed of a relatively small amount for the inserted software instructions and a larger amount for data transfer. This is due to the latency of 14 accelerator clock cycles for each AXI register read or write operation.

Further, we compared our results with LegUp HLS. As LegUp has special requirements in order to perform automated HW/SW partitioning, we run their HLS compiler stand-alone on the portion of C code that was identified as accelerator by our toolflow. The resulting FSM has 3 states and runs at a maximum speed of 170 MHz as indicated by Xilinx ISE synthesis results. This comparison shows that our HLS approach requires improvement but also that even established HLS tools hardly outperform the ARM processor. In Section VI it is discussed whether single problem speedup is required at all to gain overall system speedup.

Table I
EXAMPLE ACCELERATOR EXECUTION TIMES AND CALL OVERHEAD

	Maximum Clock Rate	FSM States	Execution Time	Relative Performance
Software	666 MHz	–	$62\mu s$	1.00
Hardware	333 MHz	10	$126\mu s$	0.49
LegUp†	170 MHz	3	$74\mu s$	0.83

†Estimation based on synthesis results using Xilinx ISE

We could demonstrate the generation of a complete and correct working HW/SW implementation from plain C without user intervention. This example shows that patching on GIMPLE level is a viable approach for seamless accelerator integration.

B. MiBench

In order to demonstrate the generality of our approach we compiled MiBench [20]. This embedded benchmark suite addresses real world problems and contains code from six different application domains. For this test we considered all accelerators found and did not apply any estimation of the expected speedup.

Due to limits in the current implementation, which are further discussed in Section VI, we could not implement and run the accelerated applications. However, the results in Table II clearly prove the generality of our approach. We are able to find and synthesize a reasonable number of accelerators from unmodified code of various application domains. Furthermore, our tests demonstrate the stability of the tool flow while analyzing a large, arbitrary codebase.

Table II
ACCELERATORS SYNTHESIZED FROM MIBENCH

Application Domain	Benchmark/ Library	Application	Accelerators	Translation Units
Network	patricia		4	1
	dijkstra		2	2
Consumer	lame		5	17
	jpeg		79	49
	tiff-v3.5.49		149	44
	mad-0.14.2b		44	35
Office	libsphinx2		39	45
	ispell		1	3
	stringsearch		11	5
	ghostscript		80	51
Automotive	bitcount		2	8
	basicmath		1	5
Telecomm.	FFT		3	3
	gsm		23	24
	CRC32		1	1
Security	pgp		107	42
	sha		9	2
			Σ 550	Σ 337

VI. Results Discussion

With our results we have shown that, firstly, our approach is valid. It can generate a working application with an attached accelerator not requiring any user interaction. Secondly, we are able to find a reasonable number of accelerators in arbitrary, unmodified code of real-world applications.

Currently, our approach is limited by our basic HLS algorithm and missing support for multiple accelerators. The focus of our work is in the GCC-plugin infrastructure that allows to transparently compile C code. Advances in HLS algorithms can be either integrated later, or we could call an existing HLS solution from within the GCC infrastructure.

Supporting multiple accelerators is a precondition for the evaluation of complex application scenarios. Since our approach wraps accelerator calls by a C-function, we plan to implement a device driver that is able to handle an arbitrary number of accelerators.

While the limitations previously mentioned do not hinder basic evaluation, exploring complex application scenarios is not feasible yet. Particularly we were not able to run any of the MiBench test cases, even though the suite offers large potential for acceleration as shown in Table II.

On platforms like Zynq, in terms of speedup, one challenge remains: One has to generate hardware running on the FPGA that outperforms the highly optimized ARM core featuring, e.g., conditional instructions and out-of-order execution. We believe that increasing single-accelerator performance by improved HLS or higher clock rates is not the only way to gain overall application speedup. Instead, acceleration potentially could be achieved by increasing thread level parallelism rather than execution speed of a single task. This is a truism since processor vendors moved towards multicore architectures. Utilizing multiple accelerators simultaneously leads from pseudo-parallelism to real task level parallelism beyond the number of present CPU cores. Such a system-level solution may provide a speedup even with single accelerators running slower than the CPU. Furthermore, with dedicated accelerators, it is to expect that energy-efficiency would also increase, which has to be confirmed by future work.

VII. Conclusion

We presented a GCC-based workflow for accelerator generation and integration. It performs automatic HW/SW partitioning by synthesizing frequently executed loops to HDL. For seamless interfacing, the program code is patched on an abstract level not depending on target platform or accelerator interface. The whole process is neither demanding HDL skills from the user nor requiring knowledge about the underlying platform. The proposed workflow has been validated by implementing a working example on a Zynq platform.

Furthermore, a complex codebase has been compiled to demonstrate the generality of our toolflow. More than 500 accelerators could be generated from the sources of MiBench [20]. This complex example was not evaluated on the hardware platform due to current limitations of our workflow mentioned in Section VI.

VIII. Current and Future Work

To overcome current limitations we work on implementing memory access for accelerators using the ACP available on Zynq devices as well as on full integration of an arbitrary number of accelerators into the OS using a device driver. Finally we want to bundle generated hardware in form of bitfiles with the application binary. This allows instant loading and execution of any accelerated application.

Beyond that, future work addresses improvements in HLS, in particular integrating with existing HLS approaches, and the migration to partial reconfiguration. HLS improvements may include exploiting more GCC-internal optimizations by moving hardware generation after the last GIMPLE optimization pass.

References

[1] S. Waite and E. Oruklu, "FPGA-based traffic sign recognition for advanced driver assistance systems," in *Journal of Transportation Technologies*, ser. Vol.3 No.1, 2013, pp. 1–16.

[2] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: High-level synthesis for fpga-based processor/accelerator systems," in *ACM/SIGDA*, 2011, pp. 33–36.

[3] B. Fort, A. Canis, J. Choi, N. Calagar, R. Lian, S. Hadjis, Y. T. Chen, M. Hall, B. Syrowik, T. Czajkowski, S. Brown, and J. Anderson, "Automating the design of processor/accelerator embedded systems with legup high-level synthesis," in *EUC*, 2014, pp. 120–129.

[4] J. Huthmann, B. Liebig, J. Oppermann, and A. Koch, "Hardware/software co-compilation with the nymble system," in *ReCoSoC*, 2013, pp. 1–8.

[5] H. Lange and A. Koch, "An execution model for hardware/software compilation and its system-level realization," in *FPL*, 2007, pp. 285–292.

[6] G. Hempel, C. Hochberger, and M. Raitza, "Towards gcc-based automatic soft-core customization," in *FPL*, IEEE, Ed., 2012, pp. 687–690.

[7] R. Nane, V. Sima, C. P. Quoc, F. Goncalves, and K. Bertels, "High-level synthesis in the delft workbench hardware/software co-design toolchain," in *EUC*, 2014, pp. 138–145.

[8] Tom Feist, "Vivado Design Suite," White Paper, 2012.

[9] Berkeley Design Technology, Inc., "An Independent Evaluation of: High-Level Synthesis Tools for Xilinx FPGAs," White Paper, 2005.

[10] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in c with roccc 2.0," in *FCCM*, 2010, pp. 127–134.

[11] Mentor Graphics, "Catapult Synthesis Datasheet," documentation, 2010.

[12] S. Fernando, M. Wijtvliet, C. Nugteren, A. Kumar, and H. Corporaal, "$(AS)^2$: accelerator synthesis using algorithmic skeletons for rapid design space exploration," in *DATE*, 2015.

[13] D. Thomas, S. Fleming, G. Constantinides, and D. Ghica, "Transparent linking of compiled software and synthesized hardware," in *DATE*, 2015.

[14] N. Paulinoi, J. C. Ferreira, J. Bispo, and J. M. Cardoso, "Transparent acceleration of program execution using reconfigurable hardware," in *DATE*, 2015.

[15] S. Huang, A. Hormati, D. Bacon, and R. Rabbah, "Liquid metal: Object-oriented programming across the hardware/software boundary," in *ECOOP 2008 Object-Oriented Programming*, J. Vitek, Ed., 2008, vol. 5142, pp. 76–103.

[16] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: A java-compatible and synthesizable language for heterogeneous architectures," in *ACM*, 2010, pp. 89–108.

[17] Xilinx, "Zynq-7000 All Programmable SoC Overview," Product Spceification, 2014.

[18] Authors omitted for blind review., "Title omitted for blind review."

[19] G. Hempel, J. Hoyer, T. Pionteck, and C. Hochberger, "Register allocation for high-level synthesis of hardware accelerators targeting fpgas," in *ReCoSoC*, 2013, pp. 1–6.

[20] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, 2001, pp. 3–14.

Using System Hyper Pipelining (SHP) to Improve the Performance of a Coarse-Grained Reconfigurable Architecture (CGRA) Mapped on an FPGA

Tobias Strauch
R&D, EDAptix, Munich, Germany
tobias@edaptix.com

***Abstract*— The well known method C-Slow Retiming (CSR) can be used to automatically convert a given CPU into a multithreaded CPU with independent threads. These CPUs are then called streaming or barrel processors. System Hyper Pipelining (SHP) adds a new flexibility on top of CSR by allowing a dynamic number of threads to be executed and by enabling the threads to be stalled, bypassed and reordered. SHP is now applied on the programming elements (PE) of a coarse-grained reconfigurable architecture (CGRA). By using SHP, more performance can be achieved per PE. Fork-Join operations can be implemented on a PE using the flexibility provided by SHP to dynamically adjust the number of threads per PE. Multiple threads can share the same data locally, which greatly reduces the data traffic load on the CGRA's routing structure. The paper shows the results of a CGRA using SHP-ed RISC-V cores as PEs implemented on a FPGA.**

Keywords—System Hyper Pipelining, Symmetrical Multi-Processing, Simultaneous Multi-Threading, Coarse-Grained Reconfigurable Architecture, FPGA

I. INTRODUCTION

It takes a certain time to execute a CPU instruction. Pipelining is used to improve the execution speed of a program on a single CPU. Instruction dependencies are handled by using stall signals. C-Slow Retiming (CSR) uses pipelining to **multiply** the functionality of a CPU, automatically generating a multithreaded CPU. This is a fundamentally different outcome compared to what is known when designs are pipelined. CSR is known since the 60's and outlined by Leiserson et al. in [1]. System Hyper Pipelining (SHP) improves CSR to enable more threads to be dynamically scaled on a multithreaded CPU and fits perfectly on FPGA technologies. SHP was first introduced by Strauch in [2].

At least two common problems for Multi-Processor System-On-Chips (MPSoC), Network-on-Chips (NoCs) and coarse-grained reconfigurable architectures (CGRA) can be identified. These are software (SW) partitioning challenges and potential data routing bottlenecks. In [3], Galanis et al. show the challenges to partition critical software parts on CGRAs. Various speedups can be achieved when the right method is applied. Yongjoo et al. propose in [4] memory-aware application mapping to improve the data throughput on CGRAs. The memory bandwidth optimization is discussed in [5] by Peng et al. Performance problems become even more critical when systems are mapped on FPGAs and applications need to be executed at a certain speed.

In this paper a Hyper Pipelined Reconfigurable Architecture (HPRA) is proposed, which demos improvements of the two aforementioned problems based on a CGRA. The key technology is System Hyper Pipelining, which generates multithreaded programming elements (PE) while improving the performance per area factor at the same time. This enables for example a higher local peak performance and fork-join operations, which simplifies the software partitioning problem. It also offers local data sharing, which reduces the risk of generating data routing bottlenecks. The proposed HPRA system is compared to a CGRA which uses the same routing and PEs as the HPRA, but does not use SHP to improve the performance of the PEs. The results show how a regular processor array can benefit from using SHP. It is easy to understand, how MPSoCs and NoCs can benefit from this approach as well.

SHP is outlined in Section 1. Related work to this paper is discussed in Section 2 before the novel SHP based architecture is introduced in Section 3. Results are given in Section 4.

II. CSR AND SHP TECHNOLOGY

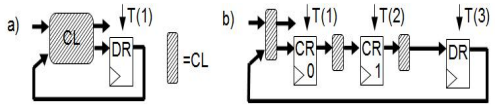


Figure 1: a) Simplified single clock design. b) Applying CSR technique.

System Hyper Pipelining (SHP) has been introduced by Strauch in [2]. This paper gives a 2-page introduction for the readers' convenience again. SHP is based on C-Slow Retiming (CSR). It enhances CSR with thread stalling, bypassing and reordering techniques by replacing the original registers of the design with memories and by adding a thread controller (TC). In the remainder of this paper, the word "thread" (T) is used synonym for the execution of a program or algorithm.

2nd International Workshop on *FPGAs for Software Programmers (FSP 2015)*, London, United Kingdom, September 1, 2015.

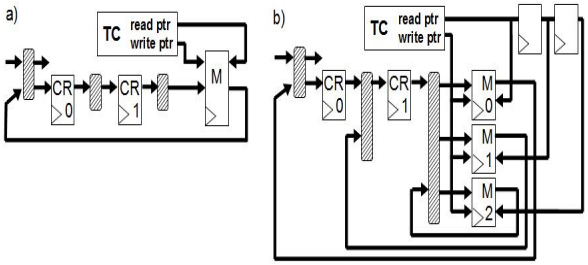


Figure 2: a) SHP-ed design with thread controller, memories and CRs. b) Further improved SHP.

Figure 1a shows the basic structure of a sequential circuit with its inputs, outputs, combinatorial logic (CL) and original design registers (DR). The sequential circuit handles one thread T(1). Figure 1b shows the CSR technique. The original logic is sliced into C (here C=3) sections. This results in C functionally independent design copies T(C=1..3) which use the logic in a time sliced fashion. Each thread has its own thread index. For each design copy it now takes C "micro-cycles" to achieve the same result as in one cycle (called "macro-cycle") of the original design. The implemented registers are called "CSR Registers", (CR) and are placed at different C-levels (CRn).

Figure 2a shows the modifications of a CSR-ed design towards SHP. Assuming the DRs are now replaced by a memory (M). The incoming design states / threads are stored at the relevant address (write pointer) based on the thread index. D is the number of threads which the memory can hold (memory depth). The outgoing thread can now be freely selected within D available threads (read pointer), except the threads already passing through the design logic. A CSR-ed design has usually many shift registers. DRs are followed by a series of CR registers. In the SHP-ed version, many memory data outputs are connected to CRs. In this case, the shift registers at the outputs can be replaced by registers at the read address inputs of the memories (Figure 2b). The memory is sliced into individual sections (M0, M1, M2) and each section has a delayed read of the thread. The outputs can now be directly connected to the relevant combinatorial logic and the shift registers can be removed. The same trick can be applied on the shift register chains at the inputs of the memory.

$$Fcsr = Forig * C * r^{C} \quad (1)$$

$$0\ Hz <= Ft <= Forig * r^{C} \quad (2)$$

$$Fshp = \Sigma\ Ft <= Fcsr \quad (3)$$

We define Forig as the maximal clock rate of the original design. The maximal speed of a CSR-ed design can be estimated by using equation 1. Fcsr is C times the original speed Forig reduced by a correction factor r^{C}, which considers the delay inserted on the critical path by the CRs. r is technology dependent. Based on empirical data, r is roughly 0.93 for a Virtex-6 FPGA and standard designs. In an SHP-ed design, a single thread can now run at any speed (over a long period) between 0 Hz (stalled) and Forig * r^{C} (Equation 2). The maximal speed of a SHP-ed design Fshp is the sum of all active threads (Equation 3). Fshp cannot be greater than Fcsr.

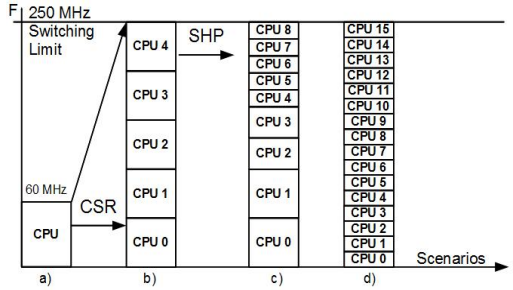


Figure 3: Histogram of different scenarios (a-d) of running CSR and SHP.

Figure 3 shows the advantages of CSR and SHP over the original design. The x-axis shows different scenarios. Assuming a single CPU runs at 60MHz on an FPGA (Figure 3a). It can be seen, how CSR improves the system performance of the original system implementation, (Figure 3b). When using CSR, the system performance is not necessarily limited by the critical path of the original design, but - for instance - by the switching limit of the FPGA (e.g. 250MHz) or the external memory access instead.

There are two key observations when SHP is used on a design. First, for executing multiple programs on multiple CPUs (symmetrical multi-processing (SMP)) or for executing multiple threads on a CPU (simultaneous multi-threading (SMT)), SHP allows a more efficient usage of the system resources. It adds the possibility to distribute the system performance over a minimum (C, Figure 3b), and a maximum (D, Figure 3c) set of design copies, whereas any solution in-between can be realized (Figure 3c). This load balancing is handled by a thread controller (TC).

Secondly, threads don't interact with each other. There is no register dependency between the individual threads. The runtime of each thread is therefore deterministic. The variable latency that the execution per thread may experience due to different behavior in if-branches for instance is not an issue, because all threads work independent of each other.

III. Basic Introduction of the Novel Architecture and Related Work

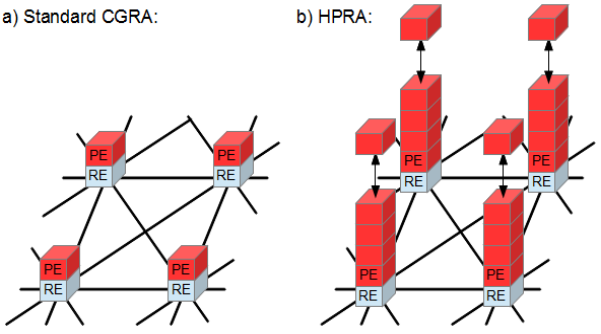


Figure 4: a) Standard CGRA with programming (PE) and routing elements (RE), b) High Performance Reconfigurable Structure with same REs but SHP-ed PEs.

Before aspects of related published work on CGRAs and NoCs are discussed, a first overview of the proposed novel architecture is given. Figure 4 gives an overview of the proposed hyper pipelined reconfigurable architecture (HPRA). The given CGRA (Figure 4a) is based on a 2 dimensional array of logic clusters, whereas each cluster has one programming element (PE) and one routing element (RE). The HPRA (Figure 4b) uses SHP to improve the performance per given area, by adding time as a 3rd dimension to generate multiple independent PE-threads which use the original PE in a time sliced fashion.

Each PE uses System Hyper Pipelining to generate multiple threads which use the logic in a time sliced fashion. One thread still runs virtually at macro-cycle (original) speed. The PE itself as well as all other elements (routing element, memory, ...) are clocked at the C times faster micro-cycle speed. Therefore the speed of the system is not necessarily dominated by the complexity of the datapath logic of the PE. The RE and the PE still run synchronously using the same clock, but it takes C micro-cycles to execute one original macro-cycle cycle of one thread.

The functional units of the application which run on the CGRA are called submodules (SM). Their individual program code is mapped to individual PEs. A PE can execute code of different SMs. Multiple threads can share the same code.

Related work must be discussed specifically with regard to the architecture's routing concept and its 3D topology. A new CGRA **routing concept** is proposed by Metzner et al. in [6]. Their proposed Quattuor-Architecture has the capability of using direct interconnects locally, within a task for fast data exchange among the submodules, and global communication using messages beyond component boundary. It tries to find an optimal trade-off between CGRA and NoC concepts. CGRAs and FPGA overlays are essentially dataflow machines to fulfill high performance requirements. A survey on CGRAs is given by Tehre et al. in [7]. Alternative concepts can be based on bus systems such as NoCs, which exchange messages among cores, memories and peripherals, all of which are connected in a network infrastructure on the chip. NoCs are usually provided as a 2-dimensional grid with routers placed at intersections between lines and columns and which are connected to homogeneous PEs. A survey on real-time NoC architectures is given by Hesham et al. in [8].

It can be said, that SHP uses time as a 3rd dimension when using the logic in a time sliced fashion. A **3D programmable logic** device has been developed by Tabula Inc. which uses operational time expansion [9] to increase performance per area. In contrast to SHP, the number of threads are fixed and individual threads cannot be stalled nor can their performance be balanced. Additionally, the logic is continuously reconfigured for each thread, which can be seen as a overhead and requires specific synthesis algorithms.

Classical **3D stacked chips** usually have problems such as that the inter-layer vias are limited in number, and the increased power density leads to high junction temperatures, as Gayasan et al. show in [10]. This interconnect bottleneck has an impact on 3D NoCs. Tradeoffs between the number of nodes utilized in the third dimension, which reduces the average number of hops traversed by a packet, and the number of physical planes used to integrate the functional blocks of the network, which decreases the length of the communication channel, is evaluated for both the latency and power consumption of a network by Pavlidis et al. in [11]. Through a detailed case study for k-ary-2-mesh networks Qian et al. have shown in [12] that transforming a 2D NoC into a 3D NoC may not improve the worst-case performance while improving the average performance.

It will be demonstrated in this paper, that SHP has a positive impact on the routing concept of a CGRA by locally sharing data, and that SHP which uses time as 3rd dimension has benefits over 3D programmable devices and stacked processor arrays.

IV. Introduction to the Hyper Pipelined Reconfigurable Architecture

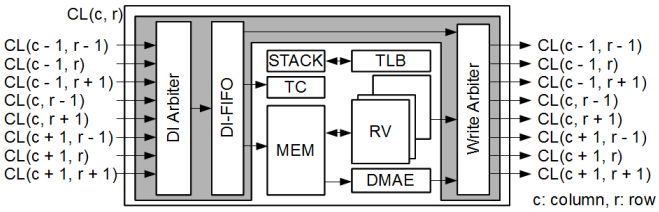


Figure 5: HPRA cluster based on routing element (gray background) and programming element.

This section discusses the hyper pipelined reconfigurable architecture (HPRA). It is based on a two dimensional array of clusters (CL), whereas each CL is based on one routing element (RE) and one programming elements (PE) as shown in Figure 5. It is also shown, how the data transfer in the system is accomplished and how the system can be partly re-configured during runtime.

A. The RISC-V based PE

The PE is based on the RISC-V (RV) instruction-set-architecture (ISA) from Berkeley [13]. The implemented 32-bit version uses a simple RISC-V subset as well as multiply and (multi-cycle) division instructions (RISC-V32IM).

With C = 4 and D = 16 the following relative performance numbers can be estimated. If less or equal to C threads are executed, then **each** thread can run at ($r^C = 0.93^4$ =) 75% speed of the original design. If the number of active threads (T) is greater than C, then the maximal system performance, which is ($C * r^C = 4 * 75\% =$) 300% of the original design, is equally distributed over all Ts.

The system is memory limited when placed on an FPGA. Therefore certain trade-offs must be considered. The RAM of each PE (PE-RAM) is dynamically shared by instructions and data. The instructions are basically a list of (independent) program sections and functions. All Ts can execute any program section or function in that RAM and can access (read/write) the complete RAM. Data can also be written by any other PE using a mechanism which is shown later.

B. Stack Handling

Each T needs its own stack range, which generates an immense memory overhead. Therefore all Ts share one single extra stack memory (PE-STACK) dynamically. A small register based translation-look-aside-buffer (TLB) uses stack access information as well as the current thread ID from the thread controller to enable access to a certain stack range. When the stack is full, a stack overflow is prevented by stalling the relevant threads until at least one other thread releases its section in the stack memory. The thread controller continuously executes all active threads, which automatically generates a round-robin mechanism for the stack usage when an overflow happens. This mechanism can still lead to a system stall in an extreme case. Therefore care must be taken when partitioning software on the individual PEs. The stack pointer register is set to 0 when a thread starts.

C. The Thread Controller

Each PE has a thread controller (TC). Each TC has special-function-registers (SFR), by which the TC can be controlled. A thread (T) can be started simply by writing the T's start address to a specific address in the TC's SFR, called "Activate". The TC then assigns the T to a specific slot (S) with has a specific slot ID (SID = {0, ..., D - 1}). If more than D threads should be started, a thread-overflow occurs. Therefore care must be taken when partitioning software on the individual PEs. A handshake mechanism must be implemented on the software layer. The task runtime can vary when more than C threads are active. Multiple threads can share the same program.

A T can "kill" itself by writing (any data) to a specific SFR, called "Exit". By doing that it frees the relevant S. A T cannot be killed by other Ts. A T can also be stalled. This means that the T's design state remains in the memory M (see section 1) and is not passed though the design logic. This allows other Ts to bypass. A T can be stalled by setting the relevant bit (=SID) to a specific SFR, called "Stall". The T starts again if this bit is cleared by any other thread. Because the SID is assigned dynamically, a certain stalling mechanism must be implemented in software. Each T can read its own SID.

To enable a fork-join program execution within one PE, the following mechanism is implemented. A set of Ts an be started from a single main T (MT) by successively writing the individual start addresses of the Ts to be started to the TC's SFR called "Activate and Count (AC)". By doing that, the number of Ts called (CT) by the MT is stored in the AC register. Optionally the MT stalls itself after that process. Each CT saves the MT's SID in the "forked thread register FT". When a CT is killed, it checks the FT and decrements the AC of the MT. If this number gets 0, the MT stalling bit is cleared by default and the MT continues. Alternatively the MT can read it's AC register to continue execution.

D. Data transfer

In the proposed architecture, each PE can run multiple threads (T). Individual Ts running on different PEs can forward data to each other using the complete HPRA's memory range. The same is true for each DMA engine, which is defined later. Data arriving at a routing element can either be forwarded to another cluster or to the PE's memory or TC. Due to the limited number of pages for this paper, this mechanism cannot be further elaborated on, but its implementation does not have a relevant impact on the achieved results shown at the end of this paper.

E. DMA Engine

To increase the system's throughput, a direct memory access engine (DMAE) is added to each PE. The DMAE has three SFR which can be programmed by each T. The DMASA register holds the start address of the source memory and the DMAL register the transfer length. The transfer is started by writing the target address of the DMA to the DMATA register. The DMAE can only be programmed when not active.

This mechanism enables a continuous datastream throughout the system. PEs connected to a system bus can initiate a burst read on the system bus. Therefore data can be read from external using bursts.

F. Configuration and partly reconfiguration during runtime

The system can easily be configured and partly reconfigured during runtime. For that instructions have to be streamed through the system to the target RAM by using the relevant target address. This can also be done during runtime so that parts of the applications can be reprogrammed / reconfigured. A thread is started by writing its start address to the relevant SFR of the TC.

V. Results

The proposed SHP based HPRA is now compared to the non SHP-ed CGRA using the same fast routing elements (RE) and the same DMA engine (DMAE). The programming element (PE) is based on a 3-stage RISC-V32IM (RV) as defined in [14]. Both designs are mapped on a Virtex 6 LX75T (-3ff784).

A. Performance per Area Improvement

Table 1 compares the data of the original CGRA and the SHP-ed HPRA. The original RV occupies 617 slices (occS) and runs at 181 MHz, which results in a performance per area factor (PpA) of 0.29 MHz/occS. The SHP-ed RV (C=4, D=16) occupies 703 slices but achieves a performance of 549 MHz. The resulting PpA is 0.78 and is 266% of the original RV's PpA. Both architectures use the same DMAE so that the programming element (PE) size difference basically results from the different RV size. The routing elements (RE) are the same as well as the system support logic (SDRAM controller and system bridge). Both have a 4x4 implementation of PE/RE clusters, whereas one cluster is removed and replace by the support logic

A higher mapping effort makes it possible that each design fits into the FPGA (maximal 11640 slices). The CGRA system with the original RV implementation achieves 2.715 GHz and a PpA of 0.23 MHz/occS. The HPRA's performance is much higher (8.235 GHz) and its PpA is 203% higher (0.77 MHz/occS) than the one of the CGRA. Thus, the HPRA can execute more threads and can achieve a 3.03-times higher

Table 1. Virtex 6 based implementation of alternative concepts.

Module	Size [occS]	Perf. [MHz]	PpA [MHz/occS]	Module	Size [occS]	Perf. [MHz]	PpA [MHz/occS]	dPpA [%]
RV	617	181	0.29	SHP-RV	703	549	0.78	266
PE	697			PE	781			
RE	103			RE	103			
Support	159			Support	159			
CGRA	**11634**	**2715**	**0.23**	**HPRA**	**11635**	**8235**	**0.77**	**303**

Table 2. Local Peak Performance Using Matrix Multiplication

	unit	4x4	5x5	6x6	7x7	8x8	9x9	10x10
RV	ns	1039	1541	2144	2845	3646	4547	5547
SHP	ns	153	202	259	330	412	505	608
Diff.	%	670	762	829	863	886	901	912

Table 3. Application Partitioning Considerations

Implementation	Threads per System	Threads per Cluster	Data Sharing per Cluster	Performance per Cluster	Performance Penalties
Single RV	15	1	no	181 MHz	
SHP-ed RV min	60 (C = 4)	4	yes	549 MHz	no
SHP-ed RV max	240 (D = 16)	16	yes	549 MHz	Yes

system performance than the non SHP-ed CGRA version on the same FPGA.

B. Local Peak Performance Improvement

In this paper, the local peak performance (LPP) defines the runtime of an algorithm based on a local data set. In other words, LPP is the execution speed of a program only accessing data available at the PE's memory and without routing data through the system. A simple matrix multiplication algorithm is used for that.

Table 2 shows the LPP for matrix multiplications of different sizes, running on the original RV implementation and a SHP-ed RV version. The SHP-ed RV can use fork-join techniques to run multiple threads in parallel. SHP allows the usage of up to D (here 16) threads. The SHP-ed RV's runtime outperforms the original version by 670% for small matrix multiplications (4x4) and 912% for larger ones (10x10).

C. Throughput Performance

It is assumed that the routing structure of a CGRA can be efficiently pipelined. The proposed system hyper pipelining technology is related to the programming element (here RISC-V32IM). Therefore, the routing structure of the two systems remains identical. Here a 4x4 system is used again, whereas 1 cluster is replaced by system logic (SDRAM interface and system bridge). This results in 15 clusters.

Table 3 gives an overview of what kind of aspects are important when an application needs to be partitioned over multiple threads. The standard CGRA system using single RV implementation can run up to 15 threads. Threads cannot share data on a cluster and the algorithm execution speed per thread is 181 MHz. The proposed HPRA system using SHP-ed RVs can run efficiently a minimum (min) of 60 threads (4 per cluster), whereas all threads on a cluster can share data locally. In this case, each thread on a cluster does not impact the runtime of the 3 remaining threads on a cluster, so that each thread gets its maximum share (137 MHz) of the 549 MHz cluster-performance. The system can be completely or locally scaled for running up to 260 threads on the same FPGA (max). 16 threads can be executed on a single cluster which can all share data locally. Here the performance of each thread is affected, because only 549 MHz are available per cluster. The number of threads per cluster can be adapted dynamically.

VI. CONCLUSION

C-Slow Retiming is a known technique to turn a digital design into a multithreaded version. System Hyper Pipelining (SHP) adds more flexibility to the multithreading approach. In this paper, SHP is applied on the programming element (PE) of a coarse-grained reconfigurable architecture (CGRA). One of the key advantages of the proposed system is a higher performance per area factor. The more concentric approach of running a flexible number of threads on a single PE improves system level aspects like local peak performance and throughput performance. Individual applications can overlap on PEs and can share data on the same RAM without moving them through the system.

In the proposed architecture (8*16=) 96 threads can communicate with a single PE, which itself can run up to 16 individual threads. The data throughput speed is higher compared to the execution time of a single instruction. Data routing and the dynamic memory access mechanism are independent of the PE's program execution. The various aspects discussed in this paper can easily be applied on MPSoCs and NoCs.

REFERENCES

[1] C. Leiserson, J. Saxe: Retiming Synchronous Circuitry. Algorithmica 6(1), 5–35 (1991)

[2] T. Strauch, "The Effects of System Hyper Pipelining on Three Computational Benchmarks Using FPGAs", 11th International Symposium in Applied Reconfigurable Computing, ARC 2015, 13-17 April 2015, Bochum, Germany, pp. 280 – 290

[3] M.D. Galanis, G. Dimitroulakos, and C. E. Goutis, “Speedups from Partitioning Critical Software Parts to Coarse-Grain Reconfigurable Hardware”, 16th IEEE Intern. Conf. on Application-Specific System

Architecture and Processors, 23-25 July 2005, Samos, Greece, pp. 50 - 55

[4] K. Yongjoo, L. Jongeun, S. Aviral, Y. Jonghee, C. Doosan, and P. Yunheung, “High Trhoughput Data Mapping for Coarse-Grained Reconfigurable Architectures”, IEEE Trans. on CAD, Vol. 30, Issue 11, Nov. 2011, pp 1599 – 1609

[5] C. Peng, J. Huiyan, L. Bo, and S. Weiwei, “Memory Bandwidth Optimizations Strategy of Coarse-Grained Reconfigurable Architecture”, Intern. Conf. on Cyber-Enabled Distributed Computing and Knowledge Discovery, 10-12 Oct. 2012, Sanya, China, pp. 228 - 231

[6] M. Metzner, J. Lizarrage, and C. Bobda, "Architecture Virtualization for Run-Time Hardware Multithreading on Field Programmable Gate Arrays”, 11th International Symposium in Applied Reconfigurable Computing, ARC 2015, 13-17 April 2015, Bochum, Germany, pp. 167 - 178

[7] V. Tehre, and R. Kshirsagar, “Survey on Coarse Grained Reconfigurable Architectures”, Intern. Journal of Computer Applications, Vol. 48, No. 16, June 2012, pp. 1 – 7

[8] S. Hesham, J. Rettkowski, D. Göhringer, and M. Abd El Ghany, “Survey on Real-Time Network-on-Chip Architectures", 11th International Symposium in Applied Reconfigurable Computing, ARC 2015, 13-17 April 2015, Bochum, Germany, pp. 191 - 202

[9] A. Rohe, S. Teig, H. Schmit, J. Redgrave, and A. Caldwell, “Operational Time Expansion“, US Patent 7587698 B1, https://www.google.tl/patents/US7236009

[10] A. Gayasen, V. Narayanan, M. Kandemir, and A. Rahman, “Designing a 3-D FPGA: Switch Box Architecture and Thermal Issues”, IEEE Trans. on VLSI, Vol. 16, No. 7, July 208, pp. 882 – 893

[11] V. Pavlidis, and E. Friedman, “3-D Topologies for Networks-on-Chip”, IEEE Trans. on VLSI, Vol. 15, No. 10, Oct. 2007, pp. 1081 - 1090

[12] Y. Qian, Z. Lu, and W. Dou, “From 2D to 3D NoCs: A Case Study on Wort-Case Communication Performance”, IEEE/ACM Intern. Conf. on Coputer-Aided Design, San Jose, CA, 2-5 Nov. 2009, pp. 555 – 562

[13] A. Waterman, Y. Lee, D. Patterson, and K Asanović, „The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0“, EECS Department, University of California, Berkeley, USA, May, 2014

Transparent hardware synthesis of Java for predictable large-scale distributed systems

Extended Abstract

Ian Gray, Yu Chan, Jamie Garside, Neil Audsley, Andy Wellings
Real-Time Systems Group, Department of Computer Science, University of York
{ian.gray, yc522, jamie.garside,
neil.audsley, andy.wellings}@york.ac.uk

***Abstract*—The JUNIPER project is developing a framework for the construction of large-scale distributed systems in which execution time bounds can be guaranteed. Part of this work involves the automatic implementation of input Java code on FPGAs, both for speed and predictability. An important focus of this work is to make the use of FPGAs transparent though runtime co-design and partial reconfiguration. Initial results show that the use of Java does not hamper hardware generation, and provides tight execution time estimates. This paper describes an overview the approach taken, and presents some preliminary results that demonstrate the promise in the technique.**

I. Introduction

Big Data is the term used for application requirements that cannot be met using existing data processing techniques, because of either the sheer scale of the input data, or the timing requirements that are placed on the system. As a result, FPGAs are starting to be deployed into data centres to exploit the large parallelism and low latency that they can offer. However, effective use of FPGAs requires significant specialist knowledge; of hardware description languages (HDLs), complex vendor tools, and high-level synthesis (HLS) systems.

As a response to this, the JUNIPER project is developing a framework for *soft real-time* Big Data systems that includes technology for automatic translation of user Java code to FPGA hardware. Rather than simply "fast enough", JUNIPER views real-time to mean that the correctness of the data is dependent on both its value and the time by which it is delivered. Hardware translation is used because hardware components have tighter bounds on their worst case response time, and are very useful for the construction of more predictable systems. Also FPGA implementations tend to display greater performance than their Java equivalents. Unlike systems that focus on using high-level synthesis to create a highly-optimised hardware implementation of a key system component, the key contribution of the JUNIPER approach is that it aims to allow totally transparent FPGA acceleration through the use of online configuration and partial dynamic reconfiguration.

The input language to the JUNIPER system is either standard Java 8, or Java written with the Real-Time Specification for Java (RTSJ). The use of Java is motivated by its common use in the large-scale data processing domain. Systems such as Hadoop are written in Java, and Spark and Storm are written partially in Java, and are implemented on the Java Virtual Machine (JVM). JUNIPER is also compatible with other JVM languages, such as Clojure and Scala.

This work has received funding from the European Union's Seventh Framework Programme under grant agreement FP7-ICT-611731

II. Programming model

The JUNIPER API is a Java 8 API for supporting large-scale computing environments, such as clusters ("cloud computing") and high-performance computers. The full details of the JUNIPER model are outside of the scope of this extended abstract and are detailed in existing work [1], [2]. In brief, JUNIPER allows the programmer to split their code into units which may be deployed into separate compute nodes. Inter-node communications, data flow, and storage, are automatically handled by the API layer.

In addition to this, JUNIPER programs can use a concept called *Locales*. Rather than placing threads and data manually using affinities, a locale is a software-level element which is used to inform the JVM and platform that the threads and data inside a locale will be tightly-coupled and so should be located as closely together as possible. These bundled threads and data items are then dynamically mapped to subsets of the target architecture, and for the purpose of this work may also be deployed to FPGAs. Online FPGA compilation and partial reconfiguration allows the system to search for a suitable mapping. This helps to solve a common problem with general-purpose acceleration of a high-level language in which it can be difficult to determine the parts of the application that should be accelerated for the largest gain.

III. Implementation strategy

The JUNIPER toolflow is shown in figure 1. The input Java (or other JVM language) is translated to C for native compilation by a real-time JVM called JamaicaVM [3]. This approach supports both standard Java and real-time Java, and allows for more predictable real-time behaviour (including real-time garbage collection). A tool called *caicos* then manages the creation of complementary hardware (FPGA) and software (host) projects. On the hardware side, the high-level synthesis tool Vivado HLS is used to translate from C to HDL.

A key requirement of this work is that use of the FPGA must be transparent to the programmer. Before the translated C can be passed to Vivado HLS, significant rewriting must be performed in order to ensure that efficient hardware is produced. First, all global memory accesses (the Java heap) from

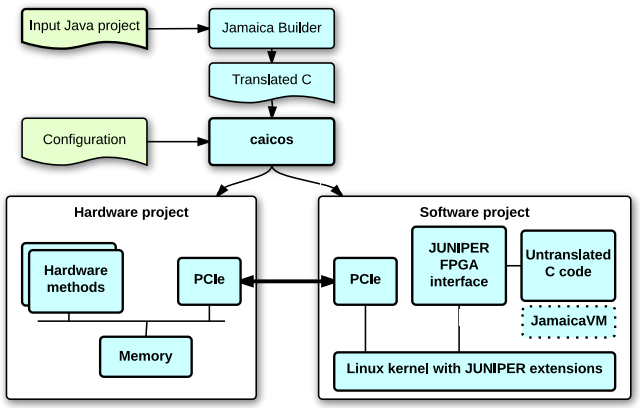


Fig. 1. The hardware and software flows in JUNIPER.

the translated C must be rewritten into AXI bus transactions. The use of pointers is avoided to ensure efficient synthesis. Secondly, because abstract or interface method calls may dynamically dispatch to different implementations based on the type of the called object, JamaicaVM and caicos perform static analysis to determine exactly which subset of methods may be called to minimise multiplexer use. Finally, untranslatable software (VM calls, native methods, etc.) are translated into a 'system call' in which the hardware calls back to the host processor over the PCIe bus to execute the required function.

The only limitation on input software is that exceptions are currently not supported inside translated methods because of the hardware complexity they introduce. It is possible to reduce this through static analysis, but this remains further work.

IV. Dynamic Acceleration

Due to space constraints on the FPGA, most of the time it will not be possible to offload all code to the FPGA simultaneously. Whilst JUNIPER allows the designer to pick a fixed subset for hardware implementation, it is also developing a dynamic acceleration approach to make the acceleration transparent to the developer through the use of online compilation, synthesis, and partial reconfiguration.

In the target domain of commercial large scale data systems, applications tend to be permanently running and can afford to dedicate a compute node to performing speculative synthesis and implementation. JUNIPER uses this to explore the design space automatically, and uses dynamic reconfiguration to swap new test bitfiles in to the running application. This is facilitated by extensive online monitoring that is provided by the JUNIPER framework. Once an improved design is found, the system will update and redeploy itself, perhaps onto fewer computer nodes if it can still guarantee its required response times.

V. Preliminary results

As this represents work in progress only relatively small filters and methods have been tested, however some interesting preliminary results have already been discovered. Table I shows comparisons between hand-written C and the JUNIPER approach (on a Xilinx Virtex 7 series device). It can be seen that the use of Java generally only imposes a small logic area and latency overhead (due to additional bus logic and memory access routines).

TABLE I. Comparison of hand-developed C and JUNIPER (naïve synthesis, without manual optimisation)

Function	Hand-developed C + HLS		Java + JamaicaVM + HLS	
	LUTs	Latency	LUTs	Latency
Vector sum	113	507	175	511
Collatz evaluation	293	278	383	282
MD5 hash	1675	3463	272	676
FIR filter	298	183	283	121

The table also shows one benefit of the approach. The MD5 result shows a huge improvement in both speed and area from using Java over C. This is because all of these numbers are before any hand-optimisation of synthesis directives. In the case of MD5, manual unrolling and function inlining can reduce the hand-developed C version to be similar in size and speed to the JamaicaVM version, but this requires specialist knowledge and is not transparent to the user. A lot of the overhead is in the C version's use of pointers, something which is removed by the restricted stack-machine of Java bytecode.

In all of these results we can see that the generated hardware has a specific latency value, rather than a range. With fixed inputs we can be certain down to the clock cycle about how long a piece of hardware will take to execute. Uncertainty can be introduced through memory latency or bus/network latency as with software implementations. These results show that there is potential for the JUNIPER acceleration approach. Evaluation of large-scale applications is currently being undertaken.

VI. Conclusion

The JUNIPER platform is an approach to building the next generation of Big Data systems which can provide design-time guarantees about their response times and performance metrics. To do this, the platform includes a range of real-time technologies, including transparent integration of FPGAs for speed and predictability.

Initial results show that the use of Java to accelerate software does not add significant overheads, and in fact when code becomes more complex and 'C-like' the JUNIPER toolflow can give better results unless manual expertise is then applied. It also provides tighter execution time estimates. This paper describes the work currently under way, the approach being developed, and presents some preliminary results that demonstrate the promise in the technique.

References

[1] I. Gray, Y. Chan, N. C. Audsley, and A. Wellings, "Architecture-awareness for real-time big data systems," in *Proceedings of the 21st European MPI Users' Group Meeting*, ser. EuroMPI/ASIA '14. New York, NY, USA: ACM, 2014, pp. 151:151–151:156.

[2] Y. Chan, A. Wellings, I. Gray, and N. Audsley, "On the locality of java 8 streams in real-time big data applications," in *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, ser. JTRES '14. New York, NY, USA: ACM, 2014, pp. 20:20–20:28.

[3] F. Siebert, "Realtime garbage collection in the jamaicavm 3.0," in *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems*, ser. JTRES '07. New York, NY, USA: ACM, 2007, pp. 94–103.