12-2016

Just In Time Assembly (JITA) - A Run Time Interpretation Approach for Achieving Productivity of Creating Custom Accelerators in FPGAs

Sen Ma
University of Arkansas, Fayetteville

Just In Time Assembly (JITA) - A Run Time Interpretation Approach for Achieving Productivity of Creating Custom Accelerators in FPGAs

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Engineering

by

Sen Ma
Capital Normal University
Bachelor of Science in Computer Engineering, 2007
Capital Normal University
Master of Science in Computer Engineering, 2010

December 2016
University of Arkansas

This dissertation is approved for recommendation to the Graduate Council

Dr. David Andrews
Dissertation Director

Dr. Christophe Bobda
Committee Member

Dr. John Gauch
Committee Member

Dr. Xuan Shi
Committee Member

Abstract

The reconfigurable computing community has yet to be successful in allowing programmers to access FPGAs through traditional software development flows. Existing barriers that prevent programmers from using FPGAs include: 1) knowledge of hardware programming models, 2) the need to work within the vendor specific CAD tools, 3) and the requirement to pass each design through synthesis, place and route.

This thesis presents a series of published papers that explore different aspects of a new approach being developed to remove the three barriers and enable programmers to compile accelerators on next generation reconfigurable manycore architectures. The approach is entitled Just In Time Assembly (JITA) of hardware accelerators. The approach has been defined to allow hardware accelerators to be built and run through software compilation and run time interpretation outside of CAD tools and without requiring each new accelerator to be synthesized. The approach advocates the use of libraries of pre-synthesized components that can be referenced through symbolic links in a similar fashion to dynamically linked software libraries. Synthesis still must occur but is moved out of the application programmers software flow and into the initial coding process that occurs when programming patterns that define a Domain Specific Language (DSL) are first coded. Programmers see no difference between creating software or hardware functionality when using the DSL. A new run time interpreter is introduced to assemble the individual pre synthesized hardware accelerators that comprise the accelerator functionality within a configurable tile array of partially reconfigurable slots at run time. Qualitative results are presented that demonstrate how software programmers can create hardware accelerators without deviating from their traditional software practices. Quantitative results are presented that compares utilization, performance, and productivity of the approach to what would be achieved by full custom accelerators created through traditional CAD flows using hardware programming models and passing through synthesis.

Acknowledgements

I would like to extend my thanks to my thesis committee, especially my advisor Dr. David Andrews for providing me the oppertunities to finish my research. I would also like to thank my parents for their support and encouragement.

Contents

List of Figures

List of Tables

List of Published Papers

This dissertation is based on the following five papers:

Chapter 2 **A Run Time Interpretation Approach For Creating Custom Accelerators**
Sen Ma, Zeyad Aklah, and David Andrews
In Proceedings of the Conference on Field-Programmable Logic and Applications (**FPL**), Sep. 2015, London, UK.

Chapter 3 **Just In Time Assembly of Accelerators**
Sen Ma, Zeyad Aklah, and David Andrews
In Proceedings of the Conference on 24th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (**FPGA**), Feb. 2016, Monterey, CA, USA.

Chapter 4 **Run Time Interpretation for Creating Custom Accelerators**
Sen Ma, Zeyad Aklah, and David Andrews
In Proceedings of the Conference on Design, Automation and Test in Europe (**DATE**), MAR. 2016, ICC, Dresden, Germany.

Chapter 5 **Breeze Computing: A Just In Time Approach for Virtualizing FPGAs in the Cloud**
Sen Ma, David Andrews, Shanyuan Gao, and Jaime Cummins
In Proceedings of the Conference on Reconfigurable Computing and FPGAs (**ReConfig**), Nov. 2016, Cancun, Mexico.

Chapter 6 **Archborn: A Custom Multiprocessor Architecture Generation Framework for Platform FPGAs**
Sen Ma, Hongyuan Ding, Miaoqing Huang, and David Andrews
In Proceedings of the Conference on Reconfigurable Computing and FPGAs (**ReConfig**), Dec. 2015, Riviera Maya, Mexico.

Chapter 1

Introduction

We are facing a new era where power and energy efficiency have become first-class design constraints within data center architectures. The Computing Research Association (CRA) working group report, entitled “Revitalizing Computer Architecture Research”, deemed this as a grand challenge problem in their “System 2020 Vision”. They put forth the challenge of creating a new featherweight supercomputer architecture that can achieve 0.001 nJ/op [1]. This is four orders of magnitude improvement over today’s systems. Reconfigurable manycores are one approach that our semiconductor industry is pursuing towards meeting this challenge [2, 3, 4].

Reconfigurable manycores are a new hybrid architecture that include a Field Programmable Gate Array (FPGA) as a coprocessor along with a traditional general purpose manycore chip. The FPGA offers the advantage that it that can be reconfigured on an application-by-application basis within the data center. The hope is the FPGA will be able to exploit, in an energy-efficient manner, the irregular types of parallelism that exists throughout emerging big data analytics and machine learning algorithms [5, 6, 7, 8, 9, 10]. Microsoft validated the energy and performance benefits of integrating FPGAs into data centers. They created an experimental server system called Catapult that allowed their document ranking algorithms to be offloaded into sets of FPGAs connected to standard Intel manycore processors. Using FPGAs allowed them to double the performance of their Bing search engine, but at only a 30% increase in energy [3].

There are two problems that will prevent widespread deployment of reconfigurable manycores throughout data centers from being successful. First designing and placing circuits within FPGAs requires hardware design skills. Second designs must be synthesized within Computer Aided Design (CAD) hardware design flows. This is represented in the top right of Figure 1.1. These problems are non-starters for successful deployment throughout our software

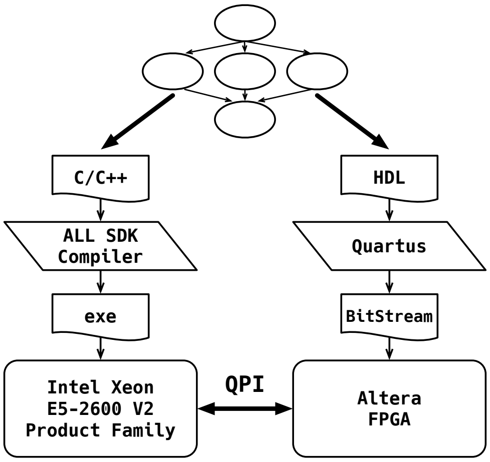


Figure 1.1: Intel Reconfigurable Manycore and QPI Interconnect.

based technology sectors. The United States Bureau of Labor Statistics reports that over 1.3M software programmers and only 85,000 hardware designers are employed within the United States [11]. This is the issue; we simply do not educate and employ sufficient numbers of hardware designers to support the coming broad deployment of these reconfigurable manycores.

1.1 Hardware Design Skills

Lack of programmer accessibility is continually called out as the biggest fundamental difficulty in taking reconfigurable computing mainstream [12]. This issue was a catalyst for the latest generation of High Level Synthesis [13]. HLS approaches start with a limited set of the C language statements and add additional constructs or pragmas to expose parallelism and guide synthesis [14, 15, 16, 17, 18, 19, 20, 21, 22, 23]. HLS and the use of imperative languages is no Panacea [24]. Generating efficient parallel circuits from sequential languages remains challenging, and does not remove the burden for programmers to study the chips underlying hardware structures. Although it is fairly easy to generate an inefficient circuit from HLS, significant effort is required to make the requisite low level platform specific optimizations to synthesize efficient circuits. Recently FPGA vendors have been working on bringing FPGAs under modern software heterogeneous programming frameworks [25, 26, 27]. These higher level

```
void(int *mem) {
  mem[512]=0;
  for(int i=0; i<512; i++) {
    mem[512] += mem[i];
  }
}
```

(a) Unoptimized: Exec. Time = 27,236 clock cycles.

```
//Width of MPort = 16 * sizeof(int)
#define ChunkSize (sizeof(MPort)/sizeof(int))
#define LoopCount (512/ChunkSize)
//Maximize data width from memory
void(MPort *mem) {
  //Use a local buffer and burst access
  MPort buff[LoopCount];
  memcpy(buff, mem, LoopCount);
  //Use a local variable for accumulation
  int sum=0;
  for(int i=0; i<LoopCount; i++) {
    //Use additional directives
    //e.g. pipeline and unroll for parallel exec.
    #pragma PIPELINE
    for(int j=0; j<ChunkSize; j++) {
      #pragma UNROLL
      sum+=(int)(buff[i]>>j*sizeof(int)*8);
    }
  }
  mem[512]=sum;
}
```

(b) Optimized: Exec. Time = 302 clock cycles.

Figure 1.2: Comparing optimized HLS and unoptimized HLS from [28].

frameworks still use HLS underneath to then create the hardware accelerators.

Generating circuits from declarative languages has also been popular [29, 30, 31, 32, 33, 34, 35, 36, 37]. Advocates argue that declarative languages provide a better starting model of concurrency for generating parallel gates compared to HLS. Despite these articulated advantages and continued interest by researchers, declarative languages have not achieved a level competitiveness with HLS within industry.

Researchers within reconfigurable computing are acknowledging the acceptance of Domain Specific Langauges (DSLs) within the software community. DSLs in general provide a restricted set of programming patterns relevant to a particular domain. DSLs such as MATLAB, SQL, Snort, Perl, and Hadoop are commonplace within our software industry. The reconfigurable computing community is increasingly investigating DSLs as front end alternatives to HLS for programming FPGAs [38, 39, 40]. DSLs are used as another layer of abstraction between the programmer and the underlying HLS tools.

The bodies of the software programming patterns still need to transformed with pragmas based on platform specific knowledge as shown in Figure 1.2. HLS tools are entirely appropriate to transform the bodies of the patterns into synthesizeable code. However starting from a DSL allows this lower level coding to be performed once by an experienced hardware designer and then reused by all application programmers. George et. al. [40, 41] showed the benefit of this approach using programming patterns from OptiML, a DSL for machine learning [42]. Figure 4.2 shows the key OptiML programming patterns that were combined and synthesized.

1.2 Use of Vendor Specific CAD Tools and Synthesis

The second issue that will block the widespread deployment of reconfigurable manycore system throughout our data centers and software based technology sectors is the use of CAD tools and synthesis. CAD tools are provided by FPGA vendors and are not interoperable. Essentially Xilinx CAD tools do not support Altera's FPGA architectures and vice versa. This represents a challenge for creating portable and reusable designs. CAD tools used for designing and programming today's FPGAs have a direct heritage to earlier VLSI hardware design tools. In general, CAD tools reinforce the notion that optimizing peak performance is an immutable first class design constraint when creating hardware circuits. Some research has been reported on how to soften the requirement of optimizing performance to achieve better designer productivity within earlier versions of vendor proprietary CAD tool flows [43, 44]. However newer versions of the Vendor tools remove this type of circumvention. What still remains true is that CAD tools integrate the steps of synthesis, place, and route within the design flow for creating hardware circuits.

Besides the use of CAD tools, designers also need to tolerate the increasingly time-consuming hardware synthesis process. The hardware synthesis process can breakdown to

#PEs	Utilization		Hardware Synthesis Time (Min)		
	LUTs	BRAM	Synthesis	Place & Route	Total Time
1	25,009 (8.0%)	36.0 (3.5%)	25.5	10.0	35.5
2	28,919 (9.5%)	45.5 (4.4%)	33.0	11.0	44.0
4	35,449 (11.7%)	65.5 (6.4%)	50.8	13.6	64.4
8	49,328 (16.3%)	105.5 (10.2%)	79.6	16.5	96.1
16	76,926 (25.3%)	185.5 (18.0%)	159.2	22.4	181.6
32	132,274 (43.6%)	345.5 (33.54%)	379.1	64.5	443.6
64	240,778 (79.3%)	665.5 (64.61%)	1257.8	110.5	1368.3

Table 1.1: Utilization and synthesis time of MPSoC with various PEs.

several steps: synthesis, placing, routing, and generating bitstreams. Table 1.1 shows the utilization and hardware synthesis time of six typical MPSoC system with different numbers of processing elements (PEs). The hardware synthesis time increases sharply following the growth of number of PEs. If any modification happens in the design, the designer have to suffer hourly re-synthesis process, which consequently complicates debugging and verification, reduces turns per day, and prevents FPGA from widespread deployment.

1.3 Objective

It is important to remove the use of CAD tools and hardware synthesis from programmers design path. This can be addressed by exploiting:

- High level programming model abstraction: to allow programmers to design accelerator without the knowledge of hardware.
- Run time interpreter: to avoid the hardware synthesis, and to interpret the intermediate representation to the executable assemblies.
- Overlay design: to abstract the FPGA resource to run the executable assemblies during the runtime.

The objective of this dissertation is to demonstrate the feasibility of just in time assembly

(JITA) approach for software programmers to design accelerator without the using of CAD tools and hardware synthesis.

1.4 Thesis Contributions

Throughout exploration of this work, I have made the following set of contributions and have published my findings in the conferences referenced for each contribution.

- *Run Time Interpreter* We created a C implementation of the interpreter. The interpreter dynamically places and routes the programming patterns within different configurations of the overlay (In Chapter 2 [45]).
- *Partial Reconfigurable Tile Overlay* We created a new overlay that uses partial reconfiguration tiles within a word width 2D switch or 1D crossbar interconnection. We provided a scripting tool to automatically create different configurations of the overlay for a DSL and FPGA (In Chapter 3 [46], 4 [47]).
- *Platform Independent Interpreter Language* We created a set of platform independent interpreter calls that enable portability over different overlays. This represents a separation of policy from a mechanism that allows a single set of interpreter calls to be implemented by all platform specific interpreters managing different overlays (In Chapter 5 [48]).
- *Automated Overlay Generator* We provided an open-source overlay architecture generation tool based on TCL scripts on FPGAs, to unify different overlay topology with various programming models for rapid system generation (In Chapter 6 [49]).

During the course of my studies, I also investigated and published additional work in support of my contributions. The work includes memory hierarchy [50], energy efficiency [51], and automate generation tools [52] in the MPSoC system.

1.5 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 presents a prototype implemented by using the proposed design flow to address productivity issues of accelerator design in FPGA. Next, Chapter 3 focuses on the main design methodology of JITA for programmers to design accelerator in FPGA. The following Chapter 4, 5, 6 provides a detailed discussion of the proposed approach in this thesis to explore the trade-off among the productivity, utilization, and performance. Finally, Chapter 7 concludes the thesis and offers some potential future work that can follow this thesis.

References

[1] CRA. Revitalizing Computer Architecture Research.

[2] David Sheffield. IvyTown Xeon + FPGA: The HARP Program.

[3] E. Chung D. Chiou K. Constantinides J. Demme H. Esmaeilzadeh J. Fowers G. Gopal J. Gray M. Haselman S. Hauck S. Heil A. Hormati J. Kim S. Lanka J. Larus E. Peterson S. Pope A. Smith J. Thong P. Xiao A. Putnam, A. Caulfield and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24, June 2014.

[4] A. Caulfield E. Chung O. Firestein M. Haselman S. Heil K. Holohan M. Humphrey T. Juhasz P. Kaur S. Lanka D. Lo T. Massengill K. Ovtcharov M. Papamichael A. Putnam R. Seera R. Tadros J. Thong L. Woods D. Chiou D. Burger S. Alkalay, H. Angepat. Agile co-design for a reconfigurable datacenter. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 15–15, New York, NY, USA, 2016. ACM.

[5] S. Yao K. Guo B. Li E. Zhou J. Yu T. Tang N. Xu S. Song Y. Wang J. Qiu, J. Wang and H. Yang. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 26–35, New York, NY, USA, 2016. ACM.

[6] E. Ghasemi and P. Chow. Accelerating apache spark big data analysis with fpgas. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 94–94, May 2016.

[7] Z. Istvn, D. Sidler, and G. Alonso. Runtime parameterizable regular expression operators for databases. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 204–211, May 2016.

[8] R. Chen and V. K. Prasanna. Accelerating equi-join on a cpu-fpga heterogeneous platform. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 212–219, May 2016.

[9] Joo-Young Kim Jeremy Fowers Karin Strauss Eric Chung Kalin Ovtcharov, Olatunji Ruwase. Accelerating deep convolutional neural networks using specialized hardware, February 2015.

[10] M. Lavasani, H. Angepat, and D. Chiou. An fpga-based in-line accelerator for memcached. *IEEE Computer Architecture Letters*, 13(2):57–60, July 2014.

[11] http://www.bls.gov/ooh/Computer-and Information-Technology/. Occupational Outlook Handbook.

[12] David F. Bacon, Rodric Rabbah, and Sunil Shukla. Fpga programming for the masses. *Commun. ACM*, 56(4):56–63, April 2013.

[13] Grant Martin and Gary Smith. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009.

[14] Maya Gokhale and Ron Minnich. FPGA Computing in a Data Parallel C. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 94–101, 1994.

[15] *Optimized Generation of Data-Path from C Codes*, March 2005.

[16] N. Wirth. Hardware compilation: translating programs into circuits. *Computer*, 31(6):25–31, Jun 1998.

[17] Michael Fingeroff. *High-Level Synthesis Blue Book.* Xlibris Corporation, 2010.

[18] Walid A. Najjar, Wim Bohm, Bruce A. Draper, Jeff Hammes, Robert Rinker, J. Ross Beveridge, Monica Chawathe, and Charles Ross. High-Level Language Abstraction for Reconfigurable Computing. In *IEEE Computer*, pages 63–69, August 2003.

[19] www.impulsec.com. ImpulseC. Last accessed December 7, 2016.

[20] www.celoxica.com. Celoxica. Last accessed December 7, 2016.

[21] Vivado High-Level Synthesis. www.xilinx.com/products/design-tools/vivado/integration/esl-design.html. Last accessed December 7, 2016.

[22] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh. PRISM-II Compiler and Architecture. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pages 9–16, 1993.

[23] S. Gupta, N.D. Dutt, R.K. Gupta, and A. Nicolau. SPARK: A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations. In *International Conference on VLSI Design*, pages 461–466, January 2003.

[24] Stephen A. Edwards. The Challenges of Synthesizing Hardware from C-Like Languages. *IEEE Design and Test of Computers*, 23(5):375–386, 2006.

[25] T.S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D.P. Singh. From opencl to high-performance hardware on fpgas. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 531–534, Aug 2012.

[26] M. Owaida, N. Bellas, K. Daloukas, and C.D. Antonopoulos. Synthesis of platform architectures from opencl programs. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 186–193, May 2011.

[27] A. Papakonstantinou, K. Gururaj, J.A. Stratton, Deming Chen, J. Cong, and W.-M.W. Hwu. Fcuda: Enabling efficient compilation of cuda kernels onto fpgas. In *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, pages 35–42, July 2009.

[28] N. George, Hyoukjoong Lee, D. Novo, T. Rompf, K.J. Brown, A.K. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne. Hardware system synthesis from domain-specific languages. In

Field Programmable Logic and Applications (FPL), 2014 24th International Conference on, pages 1–8, Sept 2014.

[29] David Bacon, Rodric Rabbah, and Sunil Shukla. Fpga programming for the masses. *Queue*, 11(2):40:40–40:52, February 2013.

[30] Rishiyur Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70, June 2004.

[31] J. Bachrach, Huy Vo, B. Richards, Yunsup Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. Chisel: Constructing hardware in a scala embedded language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1212–1221, June 2012.

[32] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 174–184, New York, NY, USA, 1998. ACM.

[33] Mary Sheeran. mufp, a language for vlsi design. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 104–112, New York, NY, USA, 1984. ACM.

[34] Geraint Jones and Mary Sheeran. Circuit design in ruby. *Formal methods for VLSI design*, 1, 1990.

[35] Andy Gill. Domain-specific languages and code synthesis using haskell. *Queue*, 12(4):30:30–30:43, April 2014.

[36] David Greaves and Satnam Singh. Exploiting System-Level Concurrency Abstractions for Hardware Descriptions. *ACM Transactions on Reconfigurable Technology and Systems*, 5(N):1 thru 29, October 2008.

[37] R. Wester, C. P. R. Baaij, and J. Kuper. A two step hardware design method using cλash. In *22nd International Conference on Field Programmable Logic and Applications, FPL 2012, Oslo, Norway*, pages 181–188, USA, August 2012. IEEE Computer Society.

[38] Gordon J. Brebner and Weirong Jiang. High-speed packet processing using reconfigurable computing. *IEEE Micro*, 34(1):8–18, 2014.

[39] Peter Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. Computer generation of hardware for linear digital signal processing transforms. *ACM Trans. Des. Autom. Electron. Syst.*, 17(2):15:1–15:33, April 2012.

[40] N. George, Hyoukjoong Lee, D. Novo, T. Rompf, K.J. Brown, A.K. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne. Hardware system synthesis from domain-specific languages. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8, Sept 2014.

[41] N. George, H. Lee, D. Novo, M. Owaida, D. Andrews, K. Olukotun, and P. Ienne. Automatic support for multi-module parallelism from computational patterns. In *2015 25th*

International Conference on Field Programmable Logic and Applications (FPL), pages 1–8, Sept 2015.

[42] A.K. Sujeeth, H. Lee, K.J. Brown, T. Rompf, H. Chafi, M. Wu, AR Atreya, M. Odersky, and K. Olukotun. Optiml: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the International Conference on Machine Learning. Haifa, Israel*, 2011.

[43] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings. Rapidsmith: Do-it-yourself cad tools for xilinx fpgas. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 349–355, Sept 2011.

[44] C. Lavin, B. Nelson, and B. Hutchings. Impact of hard macro size on fpga clock rate and place/route time. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–6, Sept 2013.

[45] S. Ma, Z. Aklah, and D. Andrews. A run time interpretation approach for creating custom accelerators. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sept 2015.

[46] Sen Ma, Zeyad Aklah, and David Andrews. Just in time assembly of accelerators. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 21-23, 2016*, pages 173–178, 2016.

[47] S. Ma, Z. Aklah, and D. Andrews. Run time interpretation for creating custom accelerators. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 900–905, March 2016.

[48] S. Ma and D. Andrews. Breeze computing: A just in time approach for virtualizing fpgas in the cloud. In *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, Dec 2016.

[49] S. Ma, H. Ding, M. Huang, and D. Andrews. Archborn: an open source tool for automated generation of chip heterogeneous multiprocessor architectures. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, Dec 2015.

[50] Sen Ma, Miaoqing Huang, Eugene Cartwright, and David Andrews. Scalable memory hierarchies for embedded manycore systems. In *Proceedings of the 8th International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, ARC'12, pages 151–162, Berlin, Heidelberg, 2012. Springer-Verlag.

[51] Sen Ma and David Andrews. On energy efficiency and amdahl's law in fpga based chip heterogeneous multiprocessor systems (abstract only). In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 253–253, New York, NY, USA, 2014. ACM.

[52] H. Ding, S. Ma, M. Huang, and D. Andrews. Oogen: An automated generation tool for custom mpsoc architectures based on object-oriented programming methods. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 233–240, May 2016.

Chapter 2

A Run Time Interpretation Approach For Creating Custom Accelerators

Sen Ma, Zeyad Aklah, and David Andrews

***Abstract*—** The world of software development has the notion of just-in-time compilation, run time binary translation, and language interpretation. These dynamic run time techniques support increased code portability and designer productivity. There are no such equivalences to increase the productivity or portability of creating new hardware components within Field Programmable Gate Arrays (FPGAs). Instead, creating a new hardware component requires hardware design skills and the overhead of running through synthesis, place and route. If a change is made to even a single line of code, the synthesis, place and route steps must be repeated. In this paper we present a new approach that allows hardware accelerators to be built and run using compilation and run time interpretation. Our results show the approach can enable software programmers without any hardware skills to create hardware accelerators at productivity levels consistent with software development and compilation. The same accelerator can be compiled 100× faster than synthesis. Even though the approach is focused on productivity, our observed performance results are promising. Our initial application test cases show the same accelerator written by a software programmer and synthesized through Vivado HLS or written using our DSL and compiled within our approach achieves equivalent performance.

2.1 Introduction

Enabling software developers to apply their skills over Field Programmable Gate Arrays (FPGAs) continues to be an unreached research objective in the reconfigurable computing community. In the past our inability to reach this goal did not greatly impact the adoption of FPGAs within companies that employed hardware and system design engineers. Further the extended time to market required to program and tune an FPGA to gain peak performance was an acceptable system development cost.

Enabling software developers to apply their skills to FPGAs is now becoming critical as we are witnessing a new and exciting inflection point in their use. FPGAs are now being viewed as viable commercial off the shelf components (COTS) visible to software developers. As an example, Microsoft recently revealed Catapult, a prototype server with FPGAs targeted for use in large data centers to accelerate their Bing search engine. Intel also announced a new compute node that will integrate an FPGA with a Xeon processor. These types of announcements have the potential to accelerate the exposure of FPGAs to software programmers.

This trend is now forcing the research community to address a looming problem for the success of FPGAs within these software domains; there simply are not sufficient numbers of designers with the requisite hardware and architecture design skills needed to handle the work. The United States Bureau of Labor Statistics 2012 report showed approximately 83,000 computer hardware engineers compared to 1.3M software engineers/programmers were employed in the United States [1]. How can the research community address this imbalance?

Our approach allows programmers to compose together a set of highly tuned pre-synthesized bitstreams, or primitives, from within a software development environment. The approach does not replace the need for experienced hardware designers and existing HLS languages and tools. The approach is complementary and allows the large number of programmers to compose bitstreams into new circuits without having to resynthesize. This is achieved by creating a new abstraction layer over the bitstreams that programmers use within their standard software development environments, just as if the bitstreams were more traditional dynamically linkable run time binary executables.

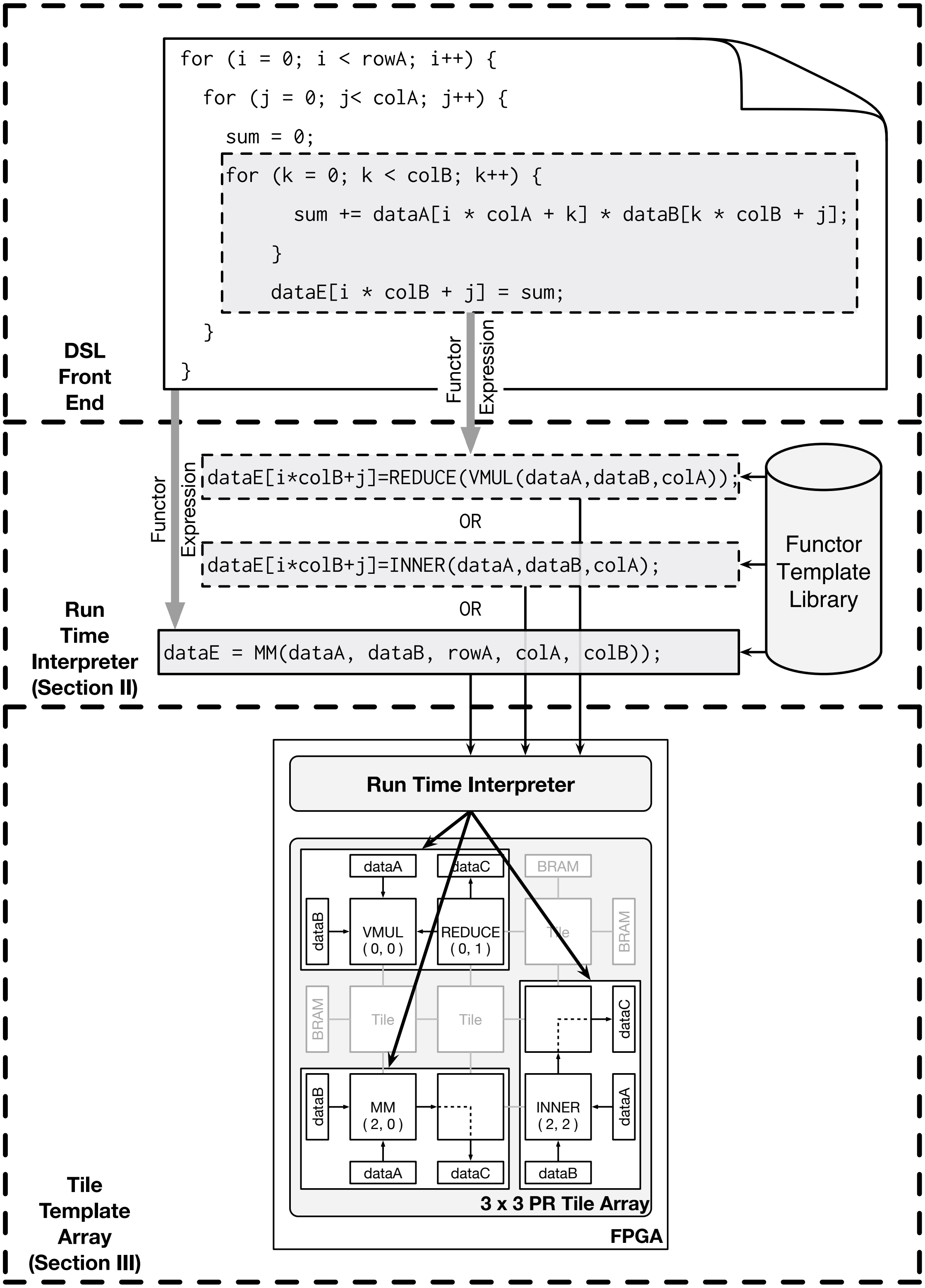
Our results validate that our objective is reachable; different accelerators can be created through compilation and composition, and interpreted at run time. Our initial results show that the same accelerator functionality that takes 14$\sim$18 minutes to synthesize in Vivado can be created by compiling and linking against an equivalent set of pre-existing bitstreams in seconds. This represents over a 100× decrease in development time.

The rest of this paper is organized as follows. The next section provides an overview of our approach. We first show how primitive operations can be combined and compiled. We then introduce a set of platform independent instructions output by our compiler. We also show how the interpreter translates and uses these instructions to configure the primitives within a 3×3 template array of partially reconfigurable slots. Section 5.5 presents our results. Section 6.5 concludes with a discussion on the additional research needed to transition the approach into commercial use.

2.2 Approach

Programmers increase their productivity by applying the fundamental software engineering tenets of abstraction, portability and reuse. They do not rewrite each new application from scratch. They take advantage of existing code bases in the form of compilable or dynamically linkable libraries. These libraries represent mechanisms that are invoked within the application code through a set of abstract policies in the form of Application Programmer Interfaces (APIs), system calls, or function prototypes. Can the same approach be applied to help programmers eliminate the need to synthesize an accelerator?

Modern FPGAs support partial reconfiguration. This allows pre-synthesized bitstreams to be downloaded from memory into the FPGA fabric at run time. We take the view that bitstreams should be treated as just another form of pre-compiled executables. The policies of these bitstreams can easily be made available to a programmer through normal software abstractions; APIs or function prototypes.



```
for (i = 0; i < rowA; i++) {
  for (j = 0; j< colA; j++) {
    sum = 0;
    for (k = 0; k < colB; k++) {
          sum += dataA[i * colA + k] * dataB[k * colB + j];
        }
        dataE[i * colB + j] = sum;
  }
}
```

```
dataE[i*colB+j]=REDUCE(VMUL(dataA,dataB,colA));
```

```
dataE[i*colB+j]=INNER(dataA,dataB,colA);
```

```
dataE = MM(dataA, dataB, rowA, colA, colB));
```

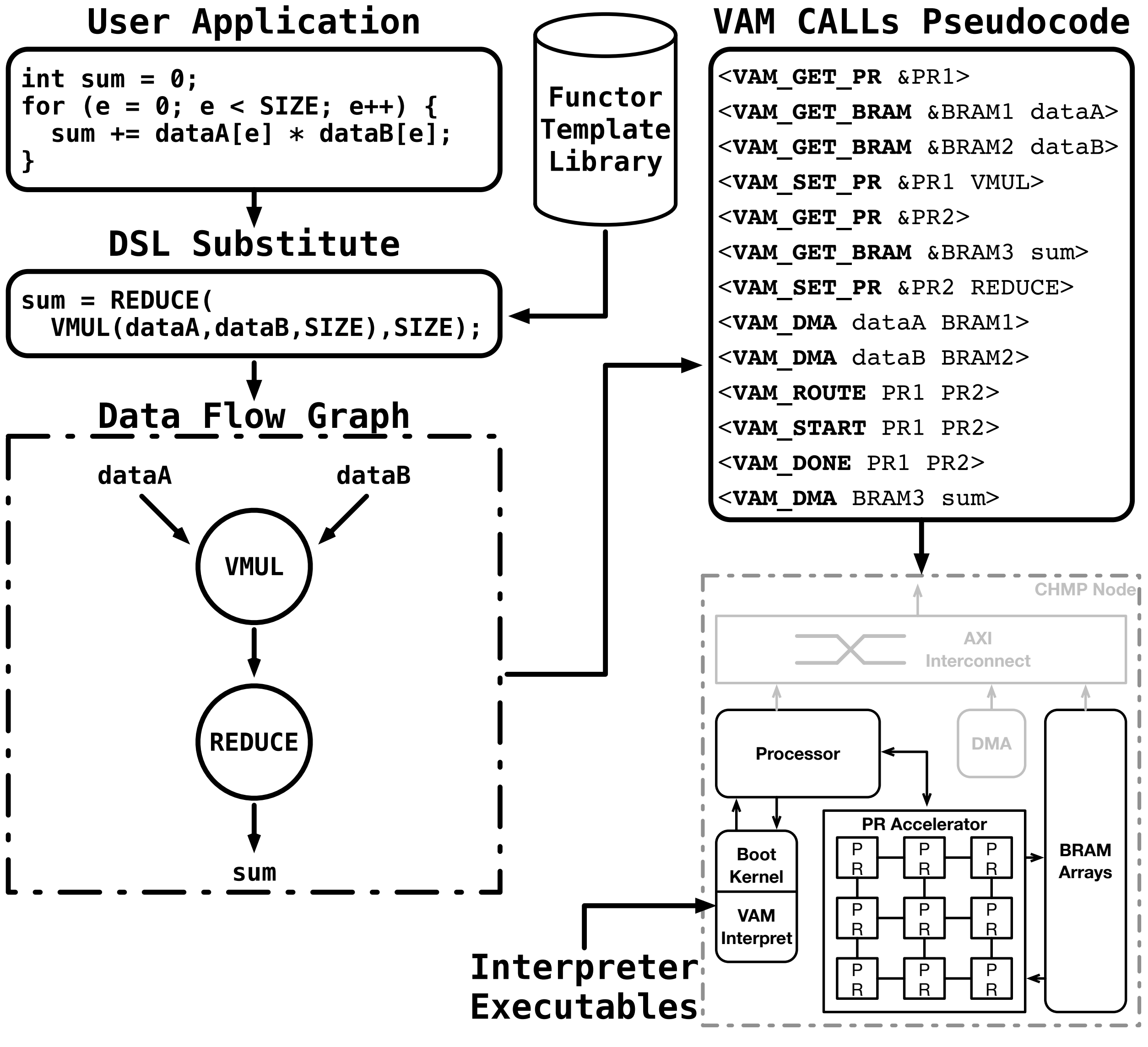
Figure 2.1: Design Flow of Matrix Multiplication Accelerator.

2.2.1 Example

Figure 2.1 shows the approach through a simple example. The code in the top of Figure 2.1 implements a simple matrix multiply that a programmer needs to turn into a hardware accelerator. Manipulating the functors can occur within a standard software development environment. We have adopted the Delite [2] framework. Delite is an existing framework developed for creating and compiling Domain Specific Languages (DSLs). The functionality of a functor is created within Delite when the DSL is defined. As part of the DSL design, each functor can be specified in C using an HLS tool, and turned into an optimized bitstream by an experienced hardware designer. It is important to note that this only occurs once during the definition of the DSL. Programmers only use the DSL after it is created. The prototypes are linked later against binaries or bitstreams within the normal compilation process.

2.2.2 Run Time Interpreter

Figure 2.1 shows a run time interpreter as part of our approach. We use the interpreter to bring portability and reuse over different organizations of reconfigurable slots. Consider the data flow graph output in Figure 6.6 produced for the inner product example. At this point we could generate code that maps each functor into a physical location for a reconfigurable slot. As an example we could map the *VMUL* functor into the co-ordinates that represent physical slot (0,0), and *REDUCE* into the co-ordinates that represent physical slot (0,1). What would happen if we changed the locations, geometries or numbers of reconfigurable slots? Using the interpreter allows the data flow graph information generated by the back end of our compiler to remain portable, similar to portable Java Byte Code. Since we target the creation of accelerators, we call this language the Virtual Accelerator Machine (VAM) language. Just as a Java Virtual Machine (JVM) provides the run time mechanisms needed to implement the policies defined by the Java Byte Code, our VAM run time interpreter provides the run time executables specific to a particular organization of partially reconfigurable slots. As a more concrete example, we created



```
int sum = 0;
for (e = 0; e < SIZE; e++) {
  sum += dataA[e] * dataB[e];
}
```

```
sum = REDUCE(
  VMUL(dataA,dataB,SIZE),SIZE);
```

```
<VAM_GET_PR &PR1>
<VAM_GET_BRAM &BRAM1 dataA>
<VAM_GET_BRAM &BRAM2 dataB>
<VAM_SET_PR &PR1 VMUL>
<VAM_GET_PR &PR2>
<VAM_GET_BRAM &BRAM3 sum>
<VAM_SET_PR &PR2 REDUCE>
<VAM_DMA dataA BRAM1>
<VAM_DMA dataB BRAM2>
<VAM_ROUTE PR1 PR2>
<VAM_START PR1 PR2>
<VAM_DONE PR1 PR2>
<VAM_DMA BRAM3 sum>
```

Figure 2.2: Compiler Flow and VAM Call Generation.

Benchmarks	Lines of Code (HLS)	Functor Composed Expression
Inner Product	35	*dataC=**REDUCE**(* ***VMUL**(dataA, dataB, Size), Size);*
$M_1 \times V_1$	60	*MatrixC=**MM**(MatrixA, VectorB, rowA, colA, 1);*
$M_1 \times M_2 \times M_3$	65	*MatrixD=**MM**(* ***MM**(MatrixA, MatrixB, rowA, colA, colB),* *MatrixC, rowA, colB, colC);*

Correlation	90	diff1=**SVSUB**(dataA, **AVG**(dataA, Size), Size);
		diff2=**SVSUB**(dataB, **AVG**(dataB, Size), Size);
		VAR1=**REDUCE**(**VSQR**(diff1, Size), SIZE);
		VAR2=**REDUCE**(**VSQR**(diff2, Size), SIZE);
		Cov=**IPR**(diff1, diff2, Size);

Table 2.1: Code Complexity

the 3×3 tile array shown in Figure 5.3 as part of our prototype system.

2.3 Experimental Results and Analysis

All evaluations were conducted using Vivado 2014.2 with corresponding Vivado HLS tools. All designs were implemented and run on a Xilinx Kintex-7 FPGA. Our test system is shown in Figure 6.6 and consisted of a MicroBlaze, AXI Interconnect, BRAM buffers, DMA, and local BRAM that held our system boot kernel and interpreter executables [3]. Reported times for synthesis and compile are averages of multiple trials. We broke our evaluation into three phases: a feasibility analysis, quantifying productivity, and quantifying performance. The objective of the feasibility analysis was to verify the practicality of the approach.

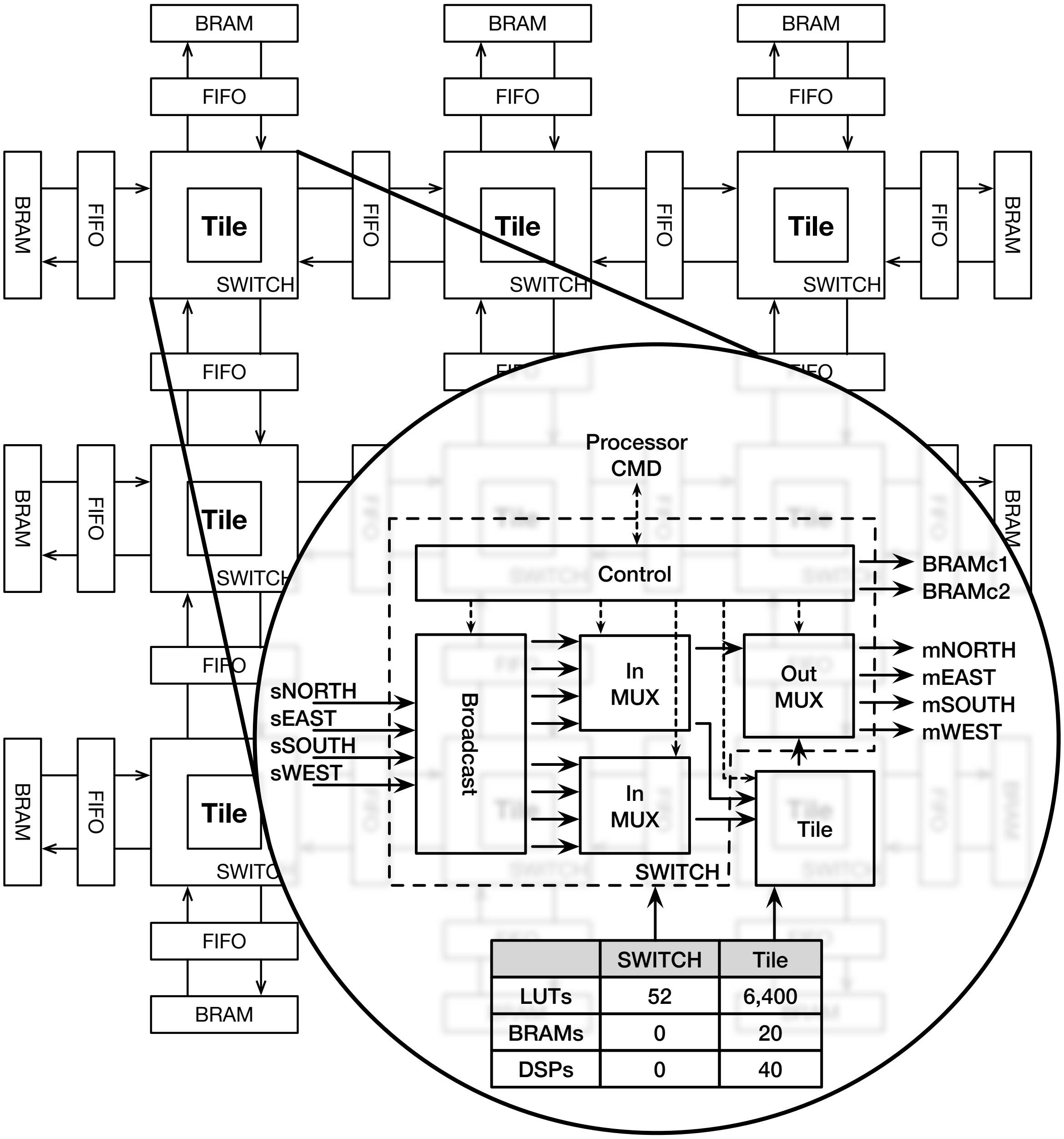
The seven functors shown in Table 4.2 offered sufficient flexibility to form a variety of test accelerators characteristic of the scientific computing and signal processing domains. The first two functors *VMUL* and *REDUCE* are familiar programming patterns. The *SVSUB*, *VSQR*, and *AVG* functors were created to form filtering patterns from signal processing. The *IPR* functor computes the inner product between two vectors. The *MM* was created to support multidimensional matrix operations from scientific computing as well as signal processing. Each functor was hand coded in C and synthesized using Vivado HLS and Vivado to create partially reconfigurable bitstreams. The functor prototypes and their bitstreams were placed in a library.

Table 4.2 shows how the functors were composed to implement the four benchmarks. The first benchmark showed how the two simple functors, *REDUCE* and *VMUL*, can be composed to create an inner product. The third benchmark showed how a three dimensional matrix multiply can be formed by composing two *MM* functors. The fourth benchmark showed how a series of five functors could be composed to form a correlation function. In all cases the functors were composed within our DSL, and all VAM interpreter calls automatically generated. The produced run time results were compared to software versions running on a workstation. All benchmarks ran successfully.

2.3.1 Productivity Analysis

Table 4.2 lists the number of C source lines written to code a single accelerator version of each benchmark. The lines of C source includes code required to interface and control the accelerator. Intuitively eliminating the need to write low level interface code will increase productivity. The thirty five lines of C code required for the inner product is replaced by composing the two *REDUCE* and *VMUL* functors. Ninety lines of C code for the correlation was replaced by five functors. We did not attempt to measure the time it took to code each benchmark accelerator. We simply use the number of source lines as a quantifying metric.

Table 3.3 quantifies productivity by comparing synthesis versus compilation times for each benchmark. Each benchmark was compiled in under $5s$. The actual time to compile just the functors was probably under a second. The reported times include additional compilation and linking that occurs within our automated system build toolchain. Our results show that compilation occurred between 170× to 214× faster than synthesis. Just this difference alone in a development flow has significant implications on programmer productivity. Under synthesis, a



	SWITCH	Tile
LUTs	52	6,400
BRAMs	0	20
DSPs	0	40

Figure 2.3: 3 × 3 Tile Array and Interconnect Network.

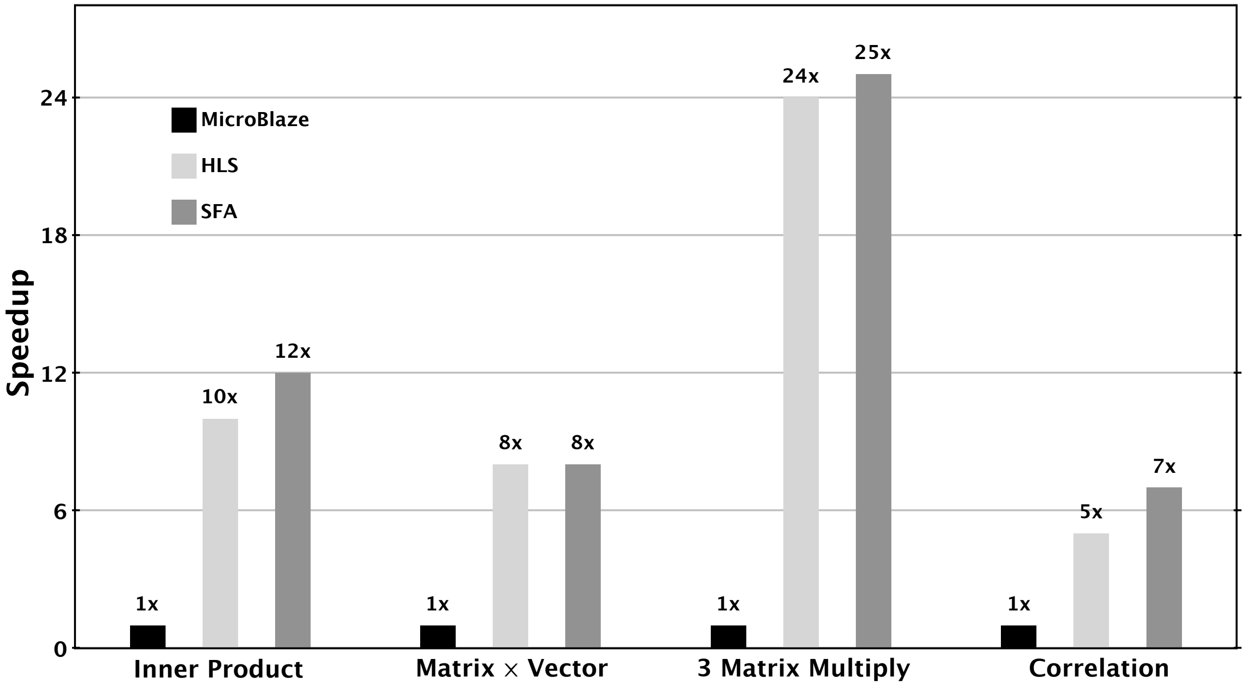


Figure 2.4: Speedup of Benchmarks.

software designer would need to wait an average of 15 minutes before the bitstreams could be generated and then tested in a running system. This equates to being able to make only four changes per hour. In comparison, a programmer using our approach can make and observe approximately 180 changes per hour. How such a level of productivity would result in rapid fix bugs is intuitive. Importantly the approach allows the programmer to quickly and easily experiment with different accelerators running within the real system.

Similar to software implementations of run time interpreters, our VAM interpreter introduced overhead. We measured an overhead of between 64μs (for the inner product) to 650μs (for the correlation) for the interpreter to select tiles, and route functors. The overheard scaled fairly linearly based on the number of tiles that were used. Although we thought it fair to call this overhead out, it should be viewed similar to the startup overhead that would be incurred once during the creation of a thread in a multithreaded operating system. After the functors have been placed and routed, this overhead is no longer seen. To be complete, there is also overhead associated with transferring bitstreams into the tiles. This overhead would also be present for

Benchmarks	SFA Compilation Overhead (s)	HLS Synthesis & PAR Overhead (s)	Productivity Improvement
Inner Product	≤ 5	865	173×
$M_1 \times V_1$	≤ 5	889	178×
$M_1 \times M_2 \times M_3$	≤ 5	1034	207×
Correlation	≤ 5	1068	214×

Table 2.2: productivity

transferring the bitstreams of the synthesized custom accelerator. In both cases the overhead is approximately equal and provides no differentiation between this approach and a traditional synthesis approach.

2.3.2 Performance Analysis

Lastly we wanted to observe if an accelerator created by composing bitstreams would suffer some measure of degraded performance compared to synthesizing a combined single bitstream version. Clearly the performance any synthesized accelerator or primitive is dependent on many different factors, including how the code is structured, the time taken to optimize the code, the designers hardware design skills. The fundamental question we sought to answer was at this stage of development would performance be degraded sufficiently to negate the approach. Figure 3.6 shows our results. We were intrigued to observe that our approach achieved speedups that were as good or better than the speedups of the synthesized accelerator. While the results are promising we do not draw any conclusions on performance based on just these prototype benchmarks. For this study we conclude that the results simply do not negate the validity of the approach. Clearly, more DSLs and more applications need to be evaluated before any real claim on performance trends can be made.

2.4 Related work

The work reported in this paper benefits from and contributes to the large body of research in programming languages and models, virtualization of hw/sw resources, operating systems, and partial reconfiguration. The use of Domain Specific Languages to generate accelerators is discussed in [4, 5, 6].

Using a DSL allows additional higher level domain specific optimizations to be made, such as fusing and transforming individual programming patterns into a more efficient representation. This optimized representation can then be run through synthesis. Our approach exploits some of the benefits of using tuned programming patterns, but differs by pre-synthesizing the patterns into functors that can be linked. We also retarget the back end to generate platform independent interpreter calls and not a synthesis tool. Research in this area roughly follows along three lines: improve and optimize the algorithms, minimize the work fed into the synthesis tool, or eliminate synthesis from the design flow by moving it into the run time system. We share commonalities with the second and third approaches. Work by Nelson [7] represents an interesting approach to minimize the work fed into the synthesis tool. Athanas combined this idea with partial reconfiguration to rapidly assemble software defined radios [8]. Lysecky and Vahid [9] explored moving synthesis from design time to run time. In their Warp Processing work, they targeted synthesizing the body of a loop while it was running. Similarly, Davor et al. [10] proposed a synthesis-free JIT approach to dynamically accelerate hot segments in a program using off line synthesized VDR units.

Our approach shares the same philosophy of creating the equivalent of macro IP components. However our approach eliminates the need to do additional routing within the physical control layer of the FPGA within the synthesis tool.

2.5 Conclusion

In this work, we presented a new approach that allows programmers to use standard software development tools and enjoy software levels of productivity when creating new hardware accelerators. The main contribution of this work is the elimination of synthesis from the path of a programmers design flow. This will greatly facilitate the use of FPGAs within our software dominated information technology sector. We validated this new approach by prototyping an end to end system running on a test system configured as a 3×3 array of tiles. The end to end system included a prototype compiler that allowed functors to be composed, type checked, and generate machine independent interpreter calls. We implemented a run time interpreter that implemented these calls on our 3×3 tile array. Our results validated the feasibility of the approach. We showed how equivalent accelerator functionality could be represented by composition, compilation and run time interpretation of functors thereby eliminating the need to pass through synthesis.

References

[1] http://www.bls.gov/ooh/Computer-and Information-Technology/. Occupational Outlook Handbook.

[2] K.J. Brown, A.K. Sujeeth, Hyouk Joong Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 89–100, Oct 2011.

[3] Sen Ma, Miaoqing Huang, and D. Andrews. Developing application-specific multiprocessor platforms on fpgas. In *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*, pages 1–6, Dec 2012.

[4] Rishiyur Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70, June 2004.

[5] N. George, Hyoukjoong Lee, D. Novo, T. Rompf, K.J. Brown, A.K. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne. Hardware system synthesis from domain-specific languages. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8, Sept 2014.

[6] O. Reiche, M. Schmid, F. Hannig, R. Membarth, and J. Teich. Code generation from a domain-specific language for c-based hls of hardware accelerators. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2014 International Conference on*, pages 1–10, Oct 2014.

[7] C. Lavin, B. Nelson, and B. Hutchings. Impact of hard macro size on fpga clock rate and place/route time. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–6, Sept 2013.

[8] A.B. MacKenzie, J.H. Reed, P. Athanas, C.W. Bostian, R.Michael Buehrer, L.A. DaSilva, S.W. Ellingson, Y.T. Hou, M. Hsiao, Jung-Min Park, C. Patterson, S. Raman, and C. da Silva. Cognitive radio and networking research at virginia tech. *Proceedings of the IEEE*, 97(4):660–688, April 2009.

[9] Roman Lysecky, Greg Stitt, and Frank Vahid. Warp processors. *ACM Transactions on Design Automation of Electronic Systems (TODAES*, 11:659–681, 2006.

[10] D. Capalija and T.S. Abdelrahman. Towards synthesis-free jit compilation to commodity fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 202–205, May 2011.

Chapter 3

Just In Time Assembly of Accelerators

Sen Ma, Zeyad Aklah, and David Andrews

Abstract— Despite the significant advancements that have been made in High Level Synthesis, the reconfigurable computing community has failed at getting programmers to use Field Programmable Gate Arrays (FPGAs). Existing barriers that prevent programmers from using FPGAs include the need to work within vendor specific CAD tools, knowledge of hardware programming models, and the requirement to pass each design through synthesis, place and route. In this paper we present a new approach that takes these barriers out of the design flows for programmers. Synthesis is eliminated from the application programmers path by becoming part of the initial coding process when creating the programming patterns that define a Domain Specific Language. Programmers see no difference between creating software or hardware functionality when using the DSL. A run time interpreter is introduced that assembles hardware accelerators within a configurable tile array of partially reconfigurable slots at run time. Initial results show the approach allows hardware accelerators to be compiled 100× faster compared to the time required to synthesize the same functionality. Initial performance results further show a compilation/interpretation approach can achieve approximately equivalent performance for matrix operations and filtering compared to synthesizing a custom accelerator.

3.1 Introduction

Just In Time (JIT) compilation and run time interpretation has been effective at delivering portability within the software world. In this work we investigate if the same JIT run time interpretation philosophy can be used to enable programmers, not hardware designers, to assemble hardware accelerators at run time on todays FPGAs.

This is no small challenge. Creating such a capability requires rethinking what is synthesized, when synthesis should occur, and how certain steps within the place and route sequence can be moved into the run time system.

Synthesis cannot be totally eliminated, but can be moved out of a programmer's compilation path. Domain Specific Languages (DSLs) provide a path forward [1, 2, 3]. Our assertion is that synthesis can be taken out of the application developers path if it is made part of the standard coding process when the Domain Specific Language (DSL) is created. This is before an application programmer attempts to combine and compile individual programming patterns into an application.

Individual programming pattern bitstreams can be made available as executable library routines. These library routines can be symbolically referred to during compilation as yet another form of a dynamically linked run time executable. Thus the bitstreams are treated no differently during compilation as traditional libraries of binary executables. The place and route steps that traditionally occur when bitstreams are combined together before synthesis can be moved into the run time system.

Java compilers produce platform independent Java bytecodes that are translated into native machine code during run time by the Java Virtual Machine (JVM). The symbolic links to the individual bitstreams can be output from a compiler as pointers to the hardware module equivalents of the native methods. They are just spatial instead of temporal representations.

The hardware native methods (bitstreams) then need to be substituted in place of the symbolic links at run time. We define a new run time interpreter to perform the equivalent function of the JVM. Instead of substituting byte codes with native machine code, the run time interpreter will substitute the symbolic links with the relocatable hardware modules.

The interpreter needs spaces within the FPGA to place the hardware modules and a programmable interconnect that can route data between the modules. FPGAs already provide the structure to support this through partial reconfiguration [4]. We create a network overlay with

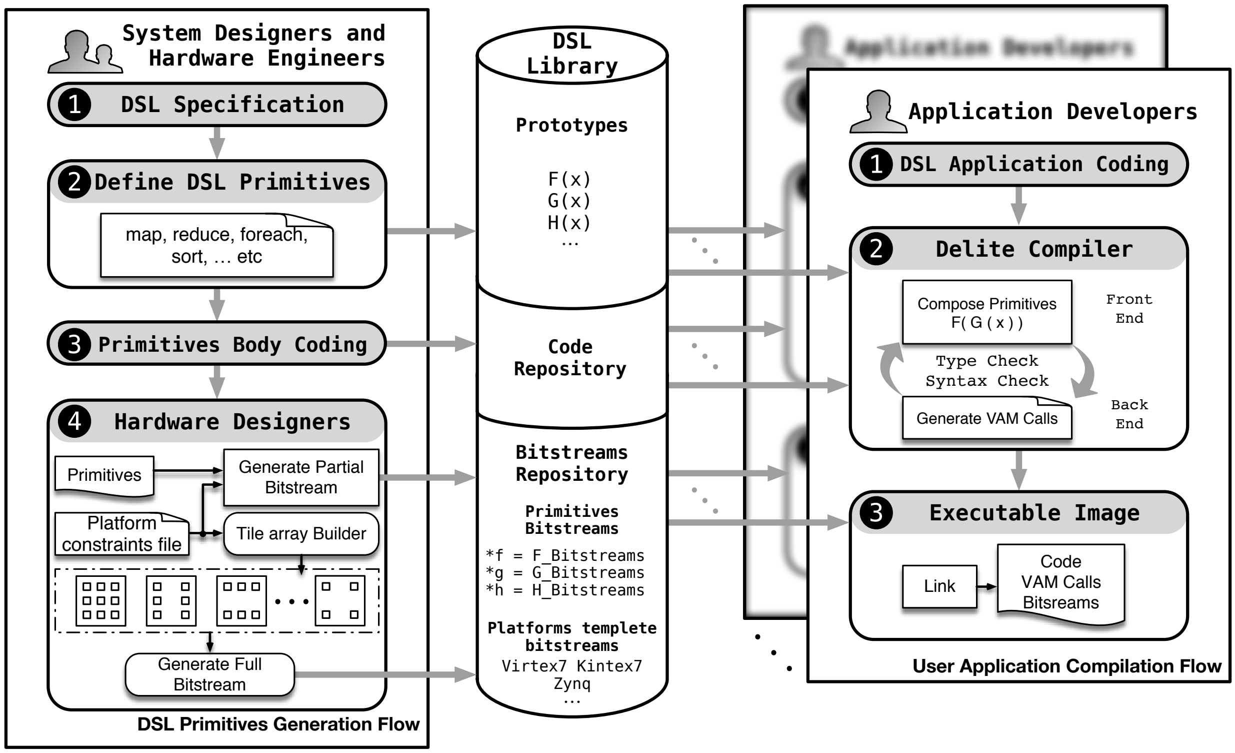


Figure 3.1: Design Flow.

programmable interconnects. Partial reconfiguration regions are provided into which the interpreter can place the modules and route them together at run time.

Figure 3.1 shows our end to end design flow. The left side shows what occurs by a system programmer when creating a Domain Specific Language (DSL) for the application programmers. During the normal coding process that occurs when programming patterns are created, we add the additional step of synthesizing each individual pattern into a bitstream. This is discussed in section 3.3. The bottom left of Figure 3.1 shows the introduction of a new type of overlay network. This overlay is created once when the programming patterns are coded and synthesized. The new overlay is discussed in section 3.2. The programming patterns, bitstreams, and overlay are placed into libraries shown in the middle of Figure 3.1 and can be accessed by all application programmers.

The application programmers use the flow on the right in Figure 3.1 to create their applications. They work with standard software DSL primitives just as if they are coding their application for a traditional software implementation. However when the programmer composes DSL primitives that have bitstream representations, the compiler inserts symbolic links to the bitstreams and builds a data flow graph representation of how the programming patterns are composed to represent the accelerator.

The compiler outputs a series of interpreter instructions that are used by an interpreter to assemble the bitstreams within the overlay and set the data connections. This is discuss in section 4.3.

3.1.1 Contributions

The main contributions of this work are:

- *PR Tile Overlay* A new overlay that uses partial reconfiguration tiles within a 2D Array and flexible word width switch boxes. The overlay represents the framework within which the run time system assembles accelerators. We provide a scripting tool to automatically create different configurations of the overlay for a DSL and FPGA.
- *Platform Independent Interpreter Language* A set of platform independent interpreter calls used by an interpreter to assemble accelerators on different configurations of the PR tile overlay at run time.
- *Run Time Interpreter* We created a C implementation of the interpreter. The interpreter dynamically places and routes the programming patterns within different configurations of the overlay. The interpreter calls are compiled and linked as sys_calls within a pthreads compliant multithreaded programming model middleware library.
- *Case Studies and Evaluation* Case studies showing a complete end to end capability. Case studies show how accelerators are formed from the programming patterns of a DSL, compiled, JIT assembled and run within different overlays. We show the portability of the approach by running the same compiled accelerator in different overlays, including single

and multiprocessor systems on chip architectures on Virtex7 and Kintex7 FPGAs. Run time performance and area overhead comparison studies are provided that compare the approach to traditional synthesis flows.

3.2 Intermediate Fabric

Commercial off-the-shelf FPGAs have served and will continue to serve as the defacto component for reconfigurable computing research. This is not because they are ideal, in fact they are far from it. Course Grained Reconfigurable Arrays (CGRAs) have been proposed as alternatives to FPGA fabrics for reconfigurable computing [5]. CGRAs replace Lookup tables (LUTs) and Flop Flops with programmable Arithmetic Logic Units (ALUs) and word width interconnects as compilation targets. CGRA structures promise to close the semantic gap between high level languages and hardware and change design flows from synthesis to compilation [6]. Even though interest in CGRAs remains high no devices are available.

Intermediate Fabrics, or overlays have been proposed that allow CGRA type structures [7] as well as more higher level computational components such as vector processors [8] to be embedded within FPGAs. The potential advantage of such overlays is that circuits and hardware acceleration can be achieved through compilation instead of synthesis on existing FPGAs. Common approaches for enabling CGRAs on an FPGA are to replace LUTs and Flip Flops with small programmable computational units such as ALUs as the compilation target. The ALUs are embedded within a network of switch boxes and channels.

The interconnect structures are defined to support wider word widths instead of bit level interconnections. The general approach introduces some overhead inefficiencies associated with need to provide additional resources to form the overlay, routing delays between the computational units, and limitations on the granularity of parallelism that can be exploited. New approaches to addressing overhead and latency issues continue to be investigated [9].

We defined a hybrid type of overlay to support JITing bitstreams.

Our overlay includes a nearest neighbor programmable word width interconnect similar to traditional CGRA type overlays. Different from traditional CGRA overlays, we expose the lookup tables and flip flops of the FPGA as partially reconfigurable tiles instead of abstracting them into programmable computational units. This combination of pre-formed interconnects and partial reconfiguration regions allows the bitstreams for the programming patterns to be downloaded at run time into the intermediate fabric. Figure 5.3 shows the structure of the hybrid overlay. The basic structure is a 2D array of partial reconfiguration tiles and programmable switches that are connected as a nearest neighbor interconnect network.

3.2.1 PR Tiles

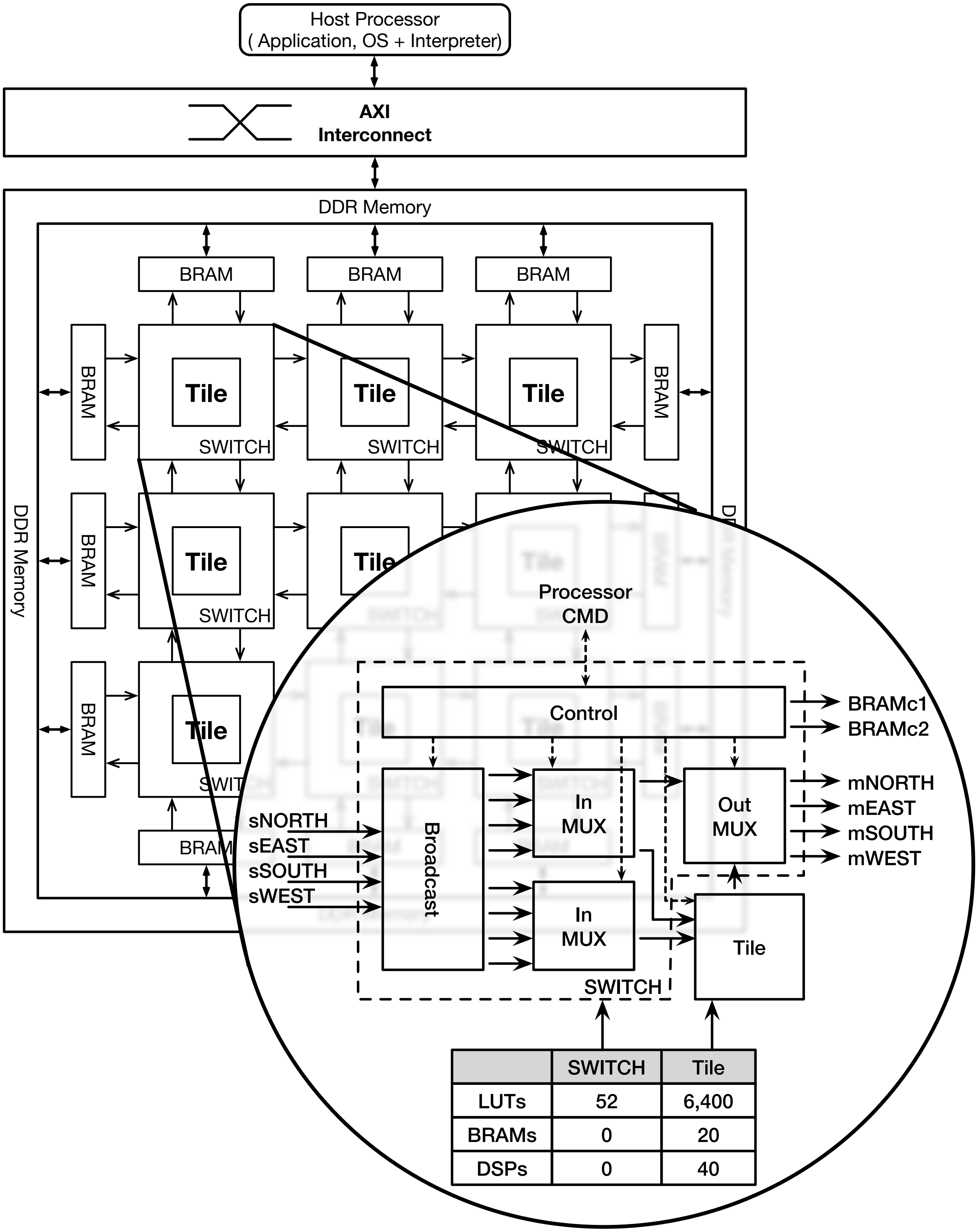
The specific 3×3 array configuration shown in Figure 5.3 was constructed of partial reconfiguration tiles sized at 9,600 LUTS, 360KB BRAM, and 80 DSPs. This particular configuration was sized to hold the largest bitstream generated from our test DSL. The exact size of the tiles is variable and can be set when the DSL is first created.

The number of the tiles is derived based on the size of the tiles and the number of resources available on a target FPGA logic family.

3.2.2 Programmable Switch

Figure 4.4 provides an exploded view of a switch. Figure 4.4 shows the types of routing patterns that can be programmed into switch. The routing patterns were defined to enable each switch to direct inputs and outputs through the tile, as well as serving as a pass through for routes between distant tiles.

Routes can be set statically or dynamically. Dynamic settings can be used for allowing the switch to support different time varying routing needs such as when multiple accelerators are



	SWITCH	Tile
LUTs	52	6,400
BRAMs	0	20
DSPs	0	40

Figure 3.2: 3 × 3 Tile Array and Interconnect Network.

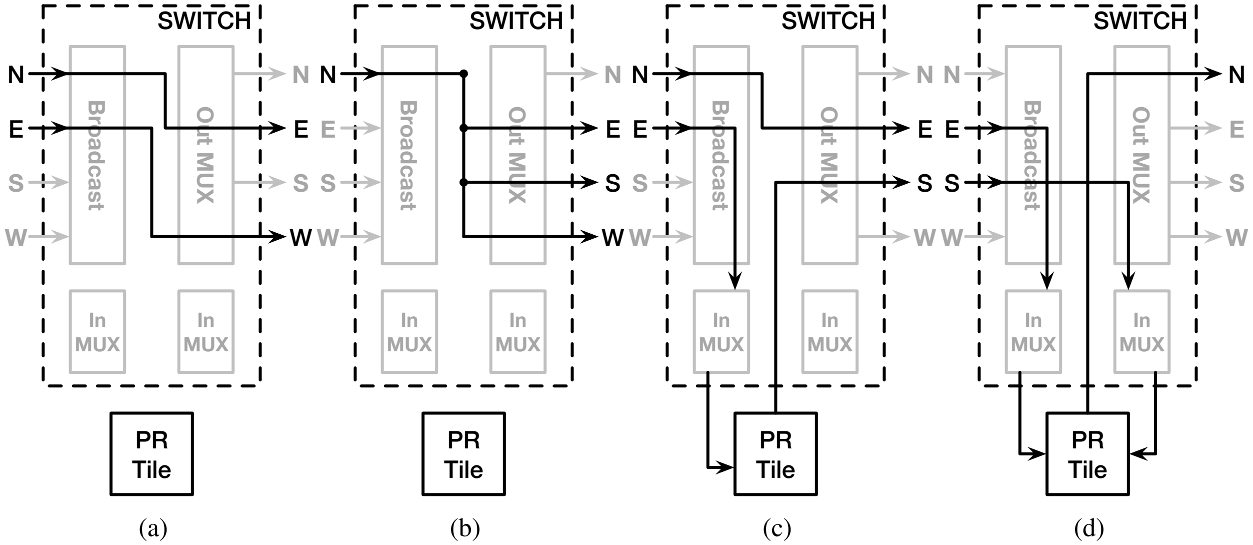


Figure 3.3: Switch Routing.

resident within the overlay. Each switch may serve as a pass through for one accelerator, and then source and synch data for a tile that is part of a different accelerator.

3.2.3 Local Memory

The boundary cells in the overlay include connections to blocks of local memories (BRAMs). These BRAMS can be used as addressable local memories or as FIFO data buffers for streaming data.

Block data transfers use DMA (not shown) between the BRAMs and Global DRAM memory. The BRAMs are placed within the global address map of the system, allowing any processor or bus master device to transfer data into and out of a local memory. The BRAMS have buffer full/empty handshaking signals that are connected through the switches to enable processing to be dynamically triggered.

3.3 The Programmers Perspective

Domain Specific Languages (DSLs) are common within software development flows and offer a reduced set of programming patterns tailored for a particular application domain.

An advantage of a DSL approach is that domain s pecific optimizations can be applied to the programming patterns before they are translated into lower level intermediate representations.

The programming patterns themselves hide complexity and allow the compiler and run time system to better decompose and map the computations across different configurations of architectures and computational resources. For these reasons, DSLs are gaining interest within the general purpose computing domain as a approach to increase performance and productivity for heterogeneous multiprocessors [2].

Our approach deviates from the current approaches that combine programming patterns prior to synthesis. When the programming patterns that define a DSL are first created, there is no reason why they cannot be synthesized before they are combined. The individual bitstream versions of the programming patterns can be placed within a corresponding library of linkable executables. The compiler can then refer to executables using symbolic links just like dynamically linked software routines.

This view presents a subtle but important difference from today's current approaches. Consider the differences in flows, design skills and development time needed to create the two separate hardware accelerators acc_1 and acc_2 using the following three generic functions $f(x), g(y), h(z)$. $acc_1 := f(g(h(z)))$; $acc_2 := g(g(h(z)))$;

Creating the first accelerator (acc_1) under current hardware design flows would require a programmer to work within a CAD tool to first combine and then synthesize the composed functionality within an HLS tool (such as Vivado). Coding each function would require the programmer to know HLS tool specific coding styles as well as board requirements. This effort typically includes inserting platform specific control, command/control and data interfaces, all requiring knowledge of low level signaling protocols. Creating the second accelerator (acc_2) would require a repeat of these steps. Under the proposed approach shown in Figure 3.1 the bitstreams for each function are created once by a hardware designer as part of the coding process for each individual function. Software prototypes can be provided for the programmers to represent the function calls. Application programmers can now compose and compile the function prototypes to create an accelerator as if they were working within a traditional software DSL framework. Assembling the individual bitstreams for $f(x), g(y), h(z)$ into a complete accelerator is moved from design time to run time.

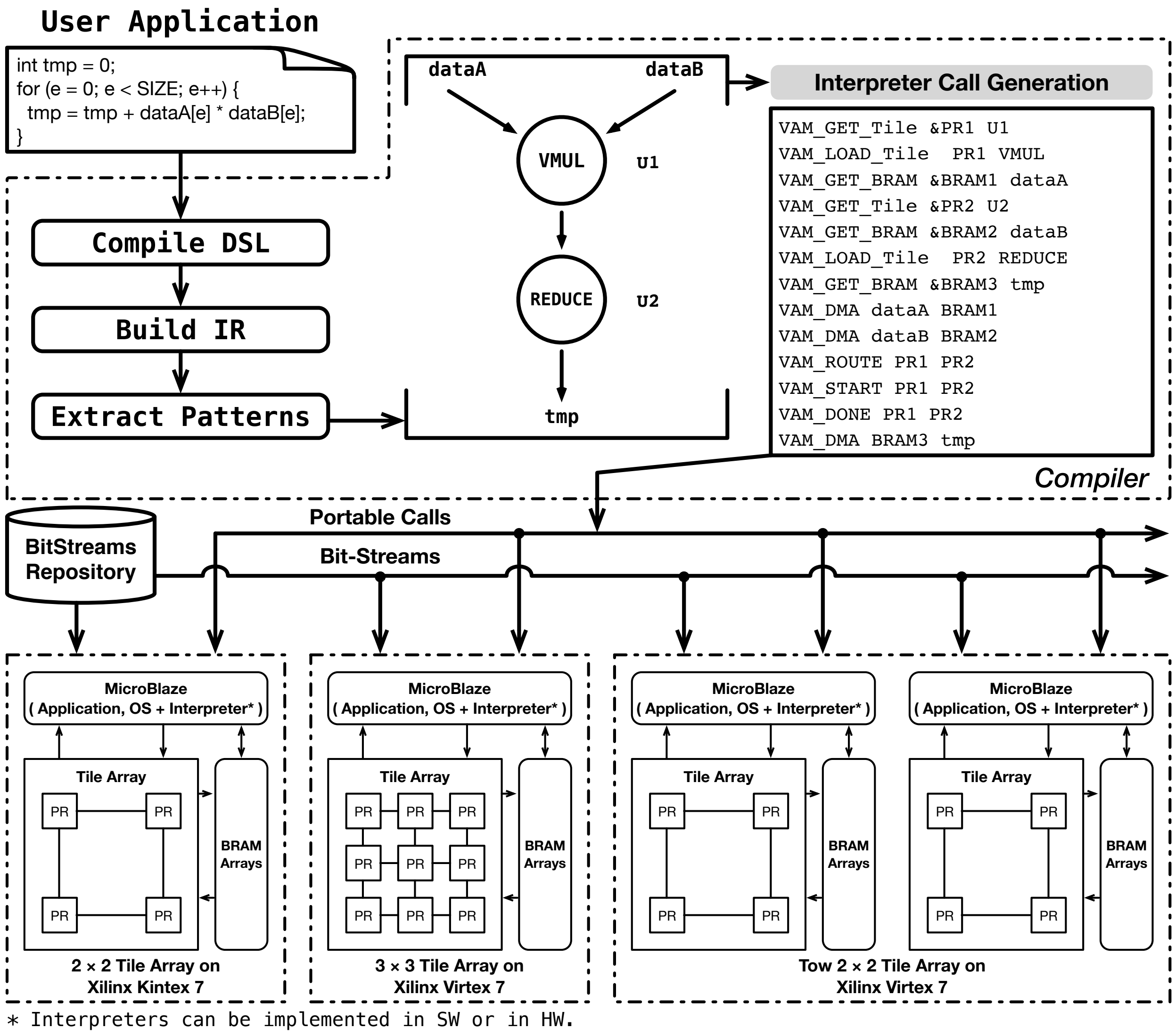
3.4 Interpreter

In this section we show how the interpreter executes the calls of the running example of Figure 4.1 on the two overlay configurations shown in Figure 5.4.

Function Placement and Loading: We chose to manage the free PR tiles in a simple queue (the VAM_TABLE). For each ***VAM_GET_TILE*** (steps 1, 4) the interpreter pops a free tile from the queue. The tiles returned for two consecutive ***VAM_GET_TILE*** calls may not be adjacent within the overlay array. This is shown in Figure 5.4. On the left the interpreter selected two adjacent tiles while on the right, the top right and bottom left tiles were selected. Function bitstreams are then loaded into free tiles using ***VAM_LOAD_TILE*** (steps 3, 6).

The run time interpreter manages input and output buffers for the accelerator in a similar fashion to tiles. For each input variable the ***VAM_GET_BRAM*** (steps 2, 5) returns a list of available local BRAMs to be selected as an input buffer.

Function Routing: After the interpreter transfers the bitstreams into the tiles, and BRAMs are selected, data paths are formed from the ***VAM_ROUTE*** (step 9) calls. For our prototype systems we implement a simplified version of the standard maze-routing algorithm [10]. The right side of Figure 5.4 shows the route formed from the top right tile (VMUL) to the bottom left



```
int tmp = 0;
for (e = 0; e < SIZE; e++) {
  tmp = tmp + dataA[e] * dataB[e];
}
```

```
VAM_GET_Tile &PR1 U1
VAM_LOAD_Tile  PR1 VMUL
VAM_GET_BRAM &BRAM1 dataA
VAM_GET_Tile &PR2 U2
VAM_GET_BRAM &BRAM2 dataB
VAM_LOAD_Tile  PR2 REDUCE
VAM_GET_BRAM &BRAM3 tmp
VAM_DMA dataA BRAM1
VAM_DMA dataB BRAM2
VAM_ROUTE PR1 PR2
VAM_START PR1 PR2
VAM_DONE PR1 PR2
VAM_DMA BRAM3 tmp
```

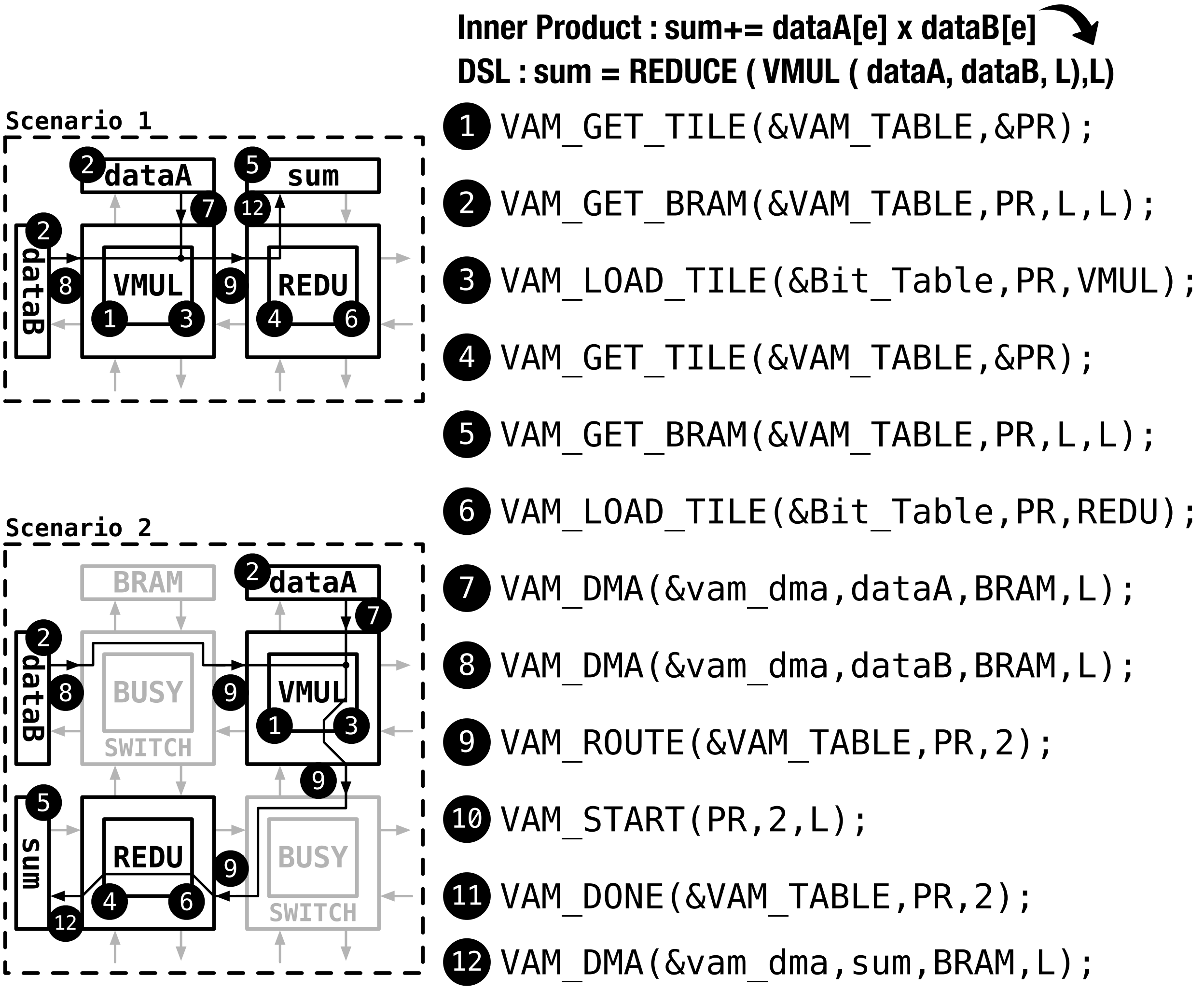
User Application
Compile DSL
Build IR
Extract Patterns
dataA
dataB
VMUL
U1
REDUCE
U2
tmp
Interpreter Call Generation
Compiler
BitStreams Repository
Portable Calls
Bit-Streams
MicroBlaze (Application, OS + Interpreter*)
Tile Array
PR
BRAM Arrays
2 × 2 Tile Array on Xilinx Kintex 7
3 × 3 Tile Array on Xilinx Virtex 7
Tow 2 × 2 Tile Array on Xilinx Virtex 7
∗ Interpreters can be implemented in SW or in HW.

Figure 3.4: Design Portability with JIT.

tile (REDU). This route first traverses down to the bottom right tile, and then left to the bottom left tile.

Data Transfer: After the accelerator has been configured, the interpreter transfers input data from DRAM into the local input buffer BRAMS using ***VAM_DMA*** (steps 7, 8). The outputs of the accelerator are transferred from the output BRAM buffer back into DRAM using ***VAM_DMA*** (step 12).

Control Operations: The ***VAM_START*** (step 10) initiates the execution of the array. The



Inner Product : sum+= dataA[e] x dataB[e]
DSL : sum = REDUCE (VMUL (dataA, dataB, L),L)

```
1 VAM_GET_TILE(&VAM_TABLE,&PR);
2 VAM_GET_BRAM(&VAM_TABLE,PR,L,L);
3 VAM_LOAD_TILE(&Bit_Table,PR,VMUL);
4 VAM_GET_TILE(&VAM_TABLE,&PR);
5 VAM_GET_BRAM(&VAM_TABLE,PR,L,L);
6 VAM_LOAD_TILE(&Bit_Table,PR,REDU);
7 VAM_DMA(&vam_dma,dataA,BRAM,L);
8 VAM_DMA(&vam_dma,dataB,BRAM,L);
9 VAM_ROUTE(&VAM_TABLE,PR,2);
10 VAM_START(PR,2,L);
11 VAM_DONE(&VAM_TABLE,PR,2);
12 VAM_DMA(&vam_dma,sum,BRAM,L);
```

Figure 3.5: Interpreter Calls For Inner Product.

VAM_DONE (step 11) returns status from the accelerator.

3.5 Experimentation and Analysis

Figure 4.1 shows the three base platform systems created to evaluate our approach. The first system contains a 2×2 array built on a Kintex7. The second contained a 3×3 overlay on a Virtex7. The third system was built on a Virtex7 and contained two 2×2 overlays. In all cases the overlays were interfaced to a MicroBlaze processor as tightly coupled accelerators.

All systems were built and synthesized using Vivado 14.2 tools.

The interpreter was written in C and cross compiled with the operating system. As all systems used a MicroBlaze we were able to compile the interpreter once and reuse it on all systems. Interpreter calls invoked through sys_calls. Sequential portions of the test programs were cross compiled and run as a thread, or in the case of mulitprocessor system, as concurrent threads on the two MicroBlazes.

3.5.1 Creating the Accelerators

The first column in Table 4.2 list the functions we selected as our benchmarks. These functions are representative of computations that a programmer might wish to accelerate from high performance computing and signal processing codes. The second column lists the lines of code that were run through our Vivado HLS tool to create custom accelerator versions of each function for comparisons.

We defined the programming patterns shown in Table 3.2 as our test DSL. Prototypes (function definitions) were created for each programming pattern, and the body of each programming pattern was coded in C as part of the DSL creation process. The C bodies were passed through Vivado HLS to generate bitstreams. We added an additional flag to the standard compilation flow to allow the C versions of the DSL to be compiled for test and evaluation, or cross compiled to run on the MicroBlaze processors for comparison. Switching the compiler flag was all that was needed to generate interpreter calls with symbolic links to the bitstreams.

The right hand column of Table 4.2 shows how we composed the programming patterns to implement the test functions. Inner product was computed using the REDUCE and VMUL programming patterns. Matrix × Vector used the matrix multiply pattern with the number of columns set to 1. Matrix × Matrix × Matrix was computed by composing two matrix multiply patterns. Correlation was computed using the five programming patterns SVSUB, AVG, REDUCE, VSQR, and IPR.

The composed expressions were compiled and our VAM call generator backend to produce

Benchmarks	Lines of Code (HLS)	Composed Expression
Inner Product	35	*dataC=*___REDUCE___*(* ___VMUL___*(dataA, dataB, Size), Size);*
$M_1 \times V_1$	60	*MatrixC=*___MM___*(MatrixA, VectorB, rowA, colA, 1);*
$M_1 \times M_2 \times M_3$	65	*MatrixD=*___MM___*(* ___MM___*(MatrixA, MatrixB, rowA, colA, colB),* *MatrixC, rowA, colB, colC);*

Correlation	90	diff1=**SVSUB**(dataA, **AVG**(dataA, Size), Size);
		diff2=**SVSUB**(dataB, **AVG**(dataB, Size), Size);
		VAR1=**REDUCE**(**VSQR**(diff1, Size), SIZE);
		VAR2=**REDUCE**(**VSQR**(diff2, Size), SIZE);
		Cov=**IPR**(diff1, diff2, Size);

Table 3.1: Code Complexity

the interpreter instructions. The run time interpreter executed the interpreter calls on each system. Qualitatively this verifies the portability of the interpreter calls over different versions of our overlay.

3.5.2 Discussion: Programmer Accessibility

Compiling is a fundamental step in getting application developers to use FPGAs; CAD tools and synthesize need to be removed from their development paths. Table 3.3 shows accelerators could be compiled and run time assembled in our overlay in less than five seconds. The reported times to compile is independent of the target overlay (2×2, 3×3) or platform (Virtex7, Kintex7). Synthesis times reported in Table 3.3 are averages. The variance in synthesis, P&R times for our test system on the Virtex7 and Kintex7 were not sufficiently different to report.

The actual time to compile just the programming patterns was more realistically under a second. The reported times include the time to compile and link the application program, middleware and operating system within our automated system build toolchain. Still the results show that compilation occurred between 170× to 214× faster than synthesis.

Patterns	Semantics	Description
REDUCE	*S = REDUCE(dataA, SizeA);*	$sum = \sum_{i=0}^{n} a_i$
VMUL	*dataC = VMUL(dataA, dataB, Size);*	$\vec{v_c} = \vec{v_a} \cdot \vec{v_b}$
MM	*MatrixC=MM(MatrixA,MatrixB,rowA,colA,colB);*	$M_c = M_a M_b$
AVG	*S = AVG(dataA, SizeA);*	$avg = (\sum_{i=0}^{n} a_i)/n$
SVSUB	*dataB = SVSUB(dataA, S, SizeA);*	$\vec{v_c} = \vec{v_a} - S$
VSQR	*dataB = VSQR(dataA, SizeA);*	$\vec{v_c} = \vec{v_a} \cdot \vec{v_a}$
IPR	*dataC = IPR(dataA, dataB, Size);*	$ipr = \sum_{i=0}^{n} a_i \times b_i$

Table 3.2: Prototype Programming Patterns

Table 3.4 shows the area cost of achieving this productivity in terms of LUTs, flop flops, and DSP blocks just for the accelerators excluding the overlay. The correlation benchmark showed the greatest increase in resources. The number of LUTS and flip flops increased 2.7× compared to synthesizing a custom version. The size of the individual programming patterns were fairly small, averaging 550 LUTS. But creating the equivalent functionality required using four of the programming patterns twice. The inner product (IPR) used two patterns, VMUL and REDUCE. Each individual pattern was smaller than the custom synthesized version, but combined resulted in a 1.9× increase in resources. This basic pattern is also present for the Matrix × Matrix × Matrix, which used the same programming pattern twice and resulted in 1.9× increase in resources. The Matrix × Vector benchmark showed approximately the same resource utilization (1.06×).

Two factors contribute to the size of the patterns. The first is the choice of programming pattern functionaltiy. These patterns were created to be general and not derived to support any one particular application. A more careful definition of pattern functionality based on application needs could eliminate using certain patterns twice. Second, no effort was put forth to optimize any of the programming patterns to reduce their footprint. The size of the programming patterns would be reduced by a skillful and careful designer. How much the sizes could be realistically reduced is unknown at this time and needs further study.

Additional resource overhead is incurred by the overlay architecture itself. Each switch required 740 LUTs. The 2×2 and 3×3 overlays required 2960, and 6660 LUTs respectively. This

Benchmarks	Compile time	HLS Synthesis & PAR (*s*	Improvement
Inner Product	≤5	865	173×
$M_1 \times V_1$	≤5	889	178×
$M_1 \times M_2 \times M_3$	≤5	1034	207×
Correlation	≤5	1068	214×

Table 3.3: productivity

is a characteristic of using an overlay. A redesign and optimization of the overlay can reduce it's resource requirements.

3.5.3 Discussion: Performance Analysis

It was anticipated that run time assembling accelerators would suffer some measure of degraded performance compared to a single custom synthesized version. We further anticipated that our initial prototypes would suffer additional performance degradations compared to later optimized revisions. Clearly the performance of any accelerator is dependent on many different factors, including how the code is structured, the time taken to optimize the code, and the designers hardware design skills. We made every attempt to apply the same types of coding style to the creation of both custom accelerators and programming patterns to eliminate any bias in comparing performance. To set a base case for comparison we also ran a software version of each benchmark on the MicroBlaze. We used the execution time of the software to compute speedups for the synthesized version and the accelerator running using our functor based approach. Figure 3.6 shows results generated on the Virtex7 on the 3×3 overlay. The results on the 2×2 array on both the Virtex7 and Kintex7 for the inner product, Matrix × Matrix × Matrix, and Matrix × Vector showed no significant differences. The Correlation benchmark required 9 programming patterns and hence 9 PR slots, so was only run on the Virtex7 3×3 overlay.

We were intrigued to observe the speedups in Figure 3.6 which showed the approach was as good or better than an equivalent synthesized custom accelerator. While the results are promising we are reluctant to draw any conclusions on performance based on these relatively few and simple benchmarks.

From a conservative perspective what we conclude is that the results simply do not negate the validity of the approach. Clearly, more DSLs and more applications need to be evaluated before any meaningful performance trends can be reported. What can be inferred is that the approach does allow a programmer to rapidly create and evaluate the execution times of accelerators. At a minimum the approach represents a powerful capability for rapidly prototyping and evaluating the performance of accelerators.

The interpreter was implemented in software as part of the operating system running on a MicroBlaze. This overhead would be seen at startup when the accelerator is assembled and does not enter into the execution time of the accelerator. In our preliminary work did run test applications to verify the ability to run time assemble different accelerators within the body of two threads running on the multiprocessor system using two MicroBlazes and two 2×2 arrays. Specifically in one thread we run time assembled the inner product benchmark and in the other a matrix multiply. The performance relationship between run time assembling and a custom accelerator is identical to the results shown in Figure 3.6 and is therefore not reported separately.

3.6 Conclusion

A new approach was presented to enable programmers to use standard software development flows to create hardware accelerators and bypass CAD tools and synthesis. The approach introduced a new PR tile overlay and set of interpreter calls that brings portability into the process. This will greatly facilitate the use of FPGAs within our software dominated information technology sector. Results were presented showing a complete end to end capability; from working within a DSL to assembling the accelerator at run time. Results also show the costs in terms of additional resource overheads for the accelerator functionality as well as the overlay.

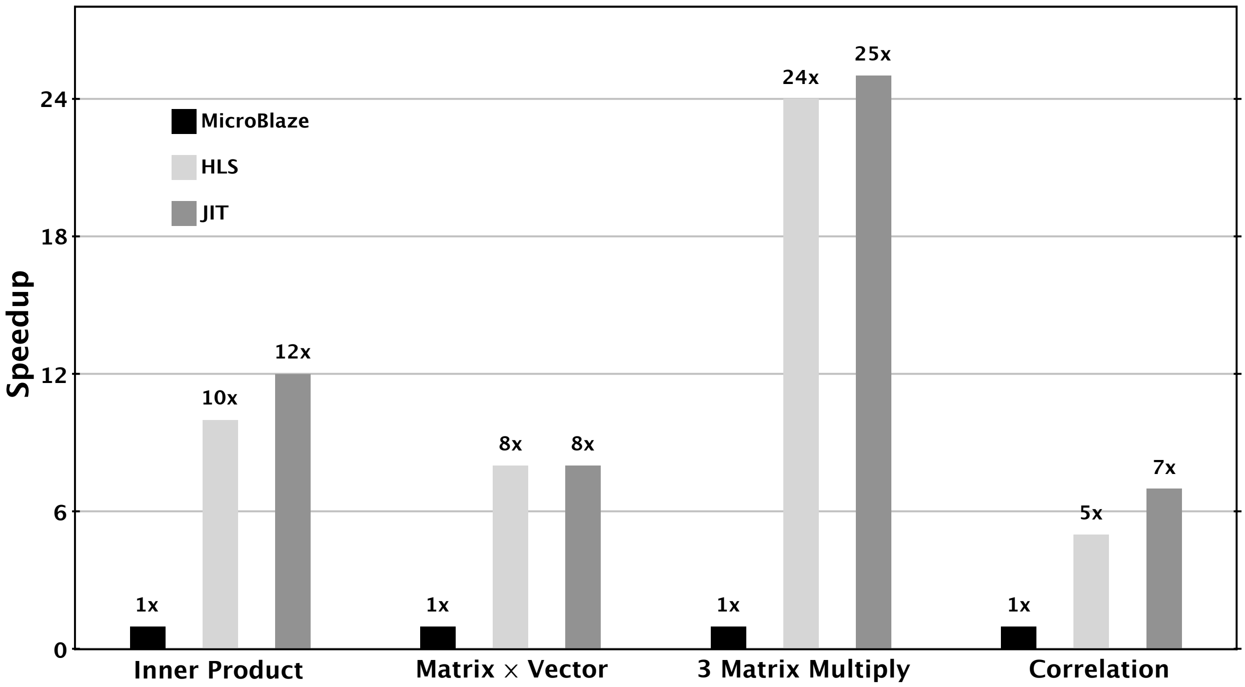


Figure 3.6: Speedup of Benchmarks.

Benchmark	Approach	Patterns	BRAM	DSP	FF	LUTs
Inner Product	JIT	VMUL	0	4	232	320
		REDUCE	0	0	167	318
		Total	0	4	399	638
	HLS		0	4	232	320
$M_1 \times V_1$	JIT	mm	24	5	848	2,254
		Total	24	5	848	2,254
	HLS		10	5	813	2,118
$M_1 \times M_2 \times M_3$	JIT	mm×2	24	5	848	2,254
		Total	48	10	1,696	4,508
	HLS		32	5	848	2,298
Correlation	JIT	AVG×2	0	2	984	2,214
		IPR	0	5	588	1,024
		SVSUB×2	0	2	427	658
		VSQR×2	0	5	588	1,024
		REDUCE×2	0	0	167	318
		Total	0	28	5,508	10,476
	HLS		32	5	2,009	3,865

Table 3.4: Resource utilization on Virtex7

References

[1] J. Bachrach, Huy Vo, B. Richards, Yunsup Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic. Chisel: Constructing hardware in a scala embedded language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1212–1221, June 2012.

[2] Hyouk Joong Lee, K.J. Brown, A.K. Sujeeth, H. Chafi, K. Olukotun, T. Rompf, and M. Odersky. Implementing domain-specific languages for heterogeneous parallel computing. *Micro, IEEE*, 31(5):42–53, Sept 2011.

[3] N. George, Hyoukjoong Lee, D. Novo, T. Rompf, K.J. Brown, A.K. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne. Hardware system synthesis from domain-specific languages. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8, Sept 2014.

[4] Sen Ma, Z. Aklah, and D. Andrews. A run time interpretation approach for creating custom accelerators. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, pages 1–4, Sept 2015.

[5] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. A fully pipelined and dynamically composable architecture of cgra. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 9–16, May 2014.

[6] João M. P. Cardoso, Pedro C. Diniz, and Markus Weinhardt. Compiling for reconfigurable computing: A survey. *ACM Comput. Surv.*, 42(4):1–65, June 2010.

[7] 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*, pages 21–24, May 2015.

[8] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. VESPA: Portable, Scalable, and Flexible FPGA-Based Vector Processors. In *CASES '08: Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 61–70, New York, NY, USA, 2008. ACM.

[9] D. Capalija and T.S. 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*, pages 1–8, Sept 2013.

[10] C.Y. Lee. An algorithm for path connections and its applications. *Electronic Computers, IRE Transactions on*, EC-10(3):346–365, Sept 1961.

Chapter 4

Run Time Interpretation for Creating Custom Accelerators

Sen Ma, Zeyad Aklah, and David Andrews

Abstract— Despite the significant advancements that have been made in High Level Synthesis, the reconfigurable computing community has not yet managed to achieve a wide-spread use of Field Programmable Gate Arrays (FPGAs) by programmers. Existing barriers that prevent programmers from using FPGAs include the need to work within vendor specific CAD tools, knowledge of hardware programming models, and the requirement to pass each design through a very time-consuming synthesis, place and route process. In this paper we present a new approach that takes these barriers out of the design flows for programmers. We move synthesis out of the programmers path and instead rely on composing pre-synthesized building blocks using a domain-specific language that supports programming patterns tailored to FPGA accelerators. Our results show that the achieved performance of run time assembling accelerators is equivalent to synthesizing a custom block of hardware using automated HLS tools.

4.1 Introduction

We are facing a new era where power and energy efficiency are first class design constraints within power hungry data centers and warehouse scale computers. The CRA working group report entitled “Revitalizing Computer Architecture Research for Next Generation Systems” called this out as a grand challenge problem for their “System 2020 Vision”. They put forth the challenge of creating a new featherweight supercomputer architecture that can achieve 0.001 nJ/op [1]. This is four orders of magnitude improvement over today’s systems.

Field Programmable Gate Arrays (FPGAs) are being viewed as an exciting new component to serve along with fixed ISA Von Neumann processors to meet the energy and performance requirements of next generation data center and warehouse scale computers. For example, Microsoft recently revealed Catapult, a server augmented with FPGAs to accelerate their Bing search engine [2]. By utilizing FPGAs, Microsoft was able to double performance at only a 30% increase in energy compared to a rack of standard processors. Interest in FPGA technology is not limited to Microsoft. Intel announced plans to integrate an FPGA with a large Xeon multi-core processor [3]. They also acquired Altera (one of the two major providers of FPGAs) for $16.7 Billion. Micron Corporation recently bought Convey and then bought Pico Computing. Convey and Pico Computing provided FPGA solutions for HPC and embedded applications. The computation fabrics available for data center architects and programmers are changing, and the age old quest of building a computer that can allow it's hardware to be tailored to perform a specific task is becoming part of this mainstream narrative.

Though FPGAs have been around for over three decades, very few FPGAs populate data centers, and even less are on acceleration boards in our PCs, and none are in our laptops and tablets. Enabling FPGAs to be part of the solution for building energy efficient next generation systems will require successfully resolving two long standing research challenges that have so far prevented reconfigurable computing from becoming mainstream.

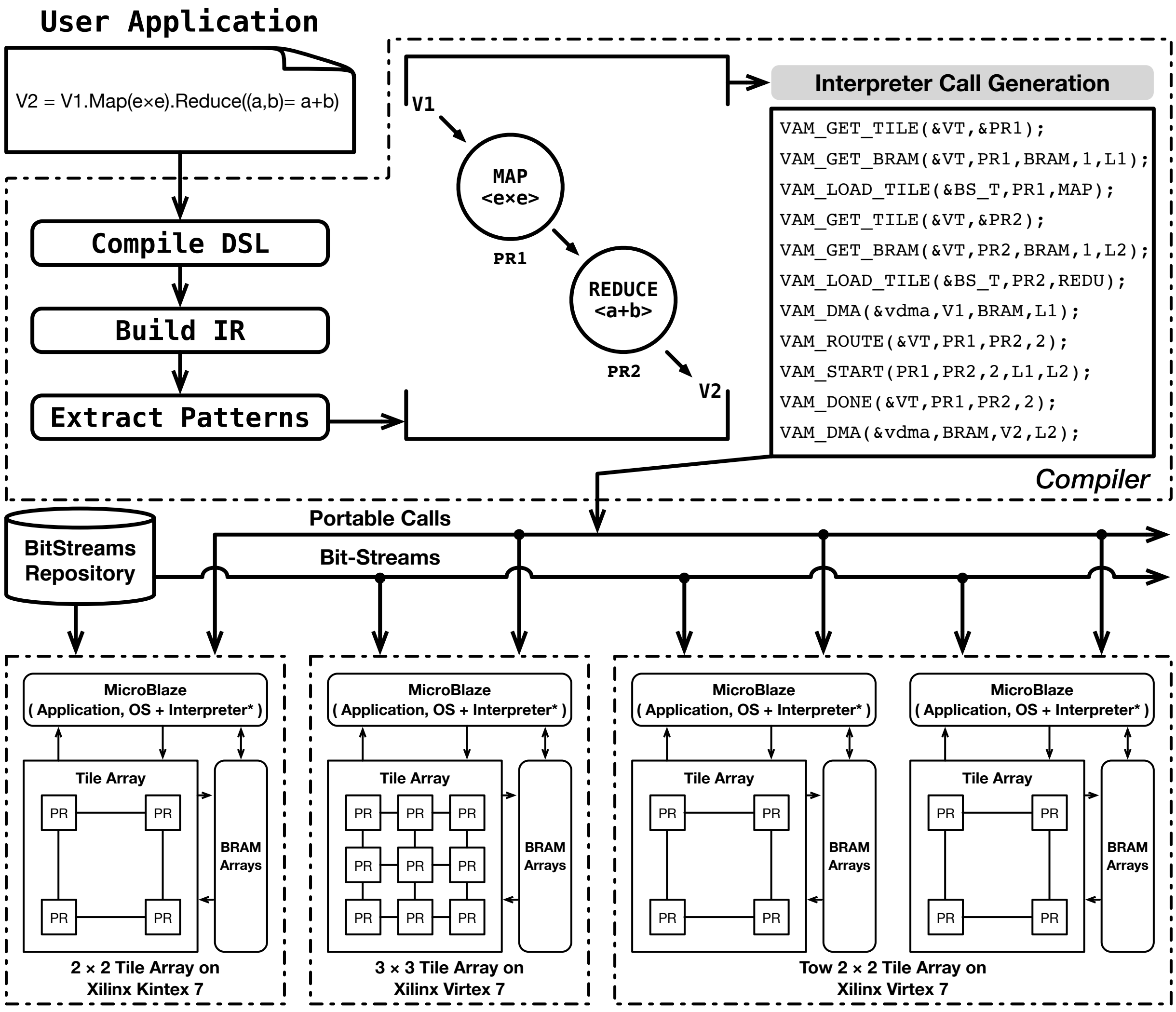
The first challenge is that the use of FPGAs still remains within the exclusive domain of hardware engineers, not software programmers. The second is the lack of a virtualization ecosystem to enable design portability and reuse. This paper presents an overview of our proposed approach to bring the JIT philosophy used to deliver portability within the software world, to enable programmers to create portable hardware accelerators. Our initial results verify that programmers can write standard software programming patterns that can be compiled and not synthesized, and produce interpreter commands that can be executed by a run time interpreter to assemble a hardware accelerator within any vendor specific FPGA.

4.1.1 Overview of Approach

Figure 4.1 motivates our approach through a simple example. At the top left of Figure 4.1 shows how a programmer would express functionality to be turned into a hardware accelerator. Domain Specific Languages (DSLs) are being used to hide complexity and promote portability within general purpose heterogeneous multiprocessor systems [4]. DSLs are also being investigated to replace High Level Synthesis languages for synthesizing circuits within FPGAs [5]. Our approach also advocates for the use of DSLs to provide programmers with platform neutral programming patterns that can be composed to express target accelerator functionality. This is shown on the top left in Figure 4.1. We provide programming patterns such as *Map* and *Reduce* which can be composed and then passed through a standard compilation process. From the programmers perspective they use the programming patterns as if they will be compiled and run on any traditional heterogeneous multiprocessor system.

Where our approach differs from earlier work to synthesize circuits from a DSL is when synthesis occurs. Synthesis for creating custom circuits within an FPGA cannot be totally eliminated. However it can be moved out of the application developers path if made part of the standard coding process of creating a Domain Specific Language (DSL). The pre-synthesized hardware representations of the programming patterns can be referenced from within a compiler as symbolic links. This is shown in the data flow graph representation of the application shown in the middle of Figure 4.1. The functions contained within the programming patterns such as e×e in *Map* and a + b in *Reduce* are not the actual code, but symbolic links to pre-synthesized bitstreams. In traditional approaches the complete application would need to be resynthesized if a single expression such as e×e is changed to log(e). The use of symbolic links allows the source program to be quickly recompiled, not resynthesized.

Our approach also differs in what the compiler outputs. Traditional DSL synthesis approaches output a platform specific hardware representation. To provide portability we re-target the compiler to output a series of platform independent interpreter instructions as shown on the



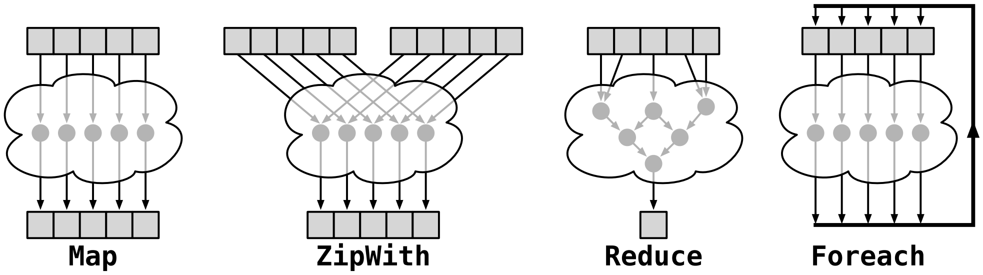
```
VAM_GET_TILE(&VT,&PR1);
VAM_GET_BRAM(&VT,PR1,BRAM,1,L1);
VAM_LOAD_TILE(&BS_T,PR1,MAP);
VAM_GET_TILE(&VT,&PR2);
VAM_GET_BRAM(&VT,PR2,BRAM,1,L2);
VAM_LOAD_TILE(&BS_T,PR2,REDU);
VAM_DMA(&vdma,V1,BRAM,L1);
VAM_ROUTE(&VT,PR1,PR2,2);
VAM_START(PR1,PR2,2,L1,L2);
VAM_DONE(&VT,PR1,PR2,2);
VAM_DMA(&vdma,BRAM,V2,L2);
```

∗ Interpreters can be implemented in SW or in HW.

Figure 4.1: Design Portability with JIT.

right in Figure 4.1. The interpreter instructions are platform independent and can direct the run time systems implemented on any platform as to how to assemble and connect the individual hardware components into an accelerator. This allows the run time system to schedule hardware circuits no differently than fat binary executables. The use of an interpreter brings portability and reuse across heterogeneous systems by separating policy from mechanism. The interpreter commands are completely platform independent.

Figure 4.1 shows the interpreter running on three different FPGAs. In our systems we implement the interpreter as part of our hthreads operating system that runs on various embedded



Pattern	Example
Map	In.map { e => e + 1 }
ZipWith	InA.zipWith(InB) { (eA , eB) => eA + eB }
Reduce	In.reduce { (e1 , e2) => e1 + e2 }
Foreach	InA.foreach { e => if (e > 0) inB(e) = true}

Figure 4.2: DSL Programming Patterns [7].

processors such as the MicroBlaze. Hthreads was developed as part of our wider investigation on resolving heterogeneity issues for FPGA based Chip Heterogeneous Multiprocessor systems. A detailed description of how hthreads removes heterogeneity issues is beyond the scope of this paper. Details on this earlier work can be found in [6].

We have defined the interpreter to be general across different platform configurations. The interpreter can be implemented on a processor or in hardware as a finite state machine. Regardless of implementation method the function of the interpreter is to assemble and control the accelerator within the FPGA at run time. The one requirement our approach places on the FPGA is the ability to support the placement of bitstreams within the reconfigurable fabric at run time. FPGAs have long provided this capability through partial reconfiguration. Figure 4.1 shows a 2-D array overlay structure that provides this capability. We embedded partial reconfiguration regions within a programmable interconnect to form a scalable overlay. As shown in Figure 4.1 the exact numbers and sizes of partial reconfiguration regions as well as the interconnect geometry can be optimized for different FPGA chips and DSL requirements. The same set of interpreter commands output by the compiler are executed by all run time interpreters, which use the command to build the accelerator based on each systems specific overlay architecture.

Type	Name	Semantics	Description
PR Region Operations	VAM_GET_TILE	bool VAM_GET_TILE (vam_table_t *VAM_TABLE, int *nPR);	Requesting a free Tile.
	VAM_GET_BRAM	bool VAM_GET_BRAM (vam_table_t *VAM_TABLE, int nPR, u32 BRAMList, int nIN, int InSize, int nOUT, int OutSize);	Requesting free BRAMs.
	VAM_LOAD_TILE	bool VAM_LOAD_TILE (XHwIcap icap, vam_Bitstream_table_t *BITSTREAM_TABLE, int nPR, int nFunctor);	Load Bitstream into Tile.
Datapath Operations	VAM_DMA	bool VAM_DMA (XAxiCdma *InstancePtr, u32 SrcAddr, u32 DstAddr, int Byte_Length);	Starting DMA from the *SrcAddr* to *DstAddr* based on the *Byte_Length*.
	VAM_ROUTE	bool VAM_ROUTE (vam_table_t *VAM_TABLE, int PR[], int nPR);	Routing the nearest neighbor 2-D switch based on the data and control path.
Control Operations	VAM_START	bool VAM_START (int PR[], int nPR, int len);	Launching the accelerator in *PR* region
	VAM_DONE	bool VAM_DONE (int PR[], int nPR, int len);	Stalling until the accelerator in PR region is done.

Table 4.1: VAM Calls

4.2 Overlay

Overlays, or intermediate fabrics, are pre-formed programmable components built on top of an FPGAs lookup tables and flip-flops. Overlays can take many forms including programmable networks, ALUs, and processors [8, 9, 10, 11, 12]. Overlays of complete heterogeneous multiprocessor systems have also been created [13]. The potential advantage of any overlay is that circuits and hardware acceleration can be achieved through compilation instead of synthesis on existing FPGAs [8, 9, 10].

We created the new type of overlay shown in Figure 5.3 as the framework within which the run time system assembles the accelerator. This new overlay adopts a nearest neighbor programmable word width interconnect that is similar in intent to traditional network on chip overlays. However instead of including programmable processors within the network, we expose the lower level lookup tables and flip flops as partially reconfigurable tiles. This combination of pre-formed interconnects and partial reconfiguration regions allows the run time system to place the individual bitstreams for the programming patterns into the individual tiles, and set the network to configure the data paths from the compilers data flow graph.

The overlay is configured as a 2D array of partial reconfiguration tiles and programmable switches that are connected as a nearest neighbor interconnect network. Network interconnects contain a FIFO to support higher clock rates and data streaming between switches. The size and number of each partial reconfiguration tile is variable and can be set based on the sizes of the bitstreams that comprise the DSL as well as the resource limitations of each specific FPGA.

4.2.1 PR Tiles

The 2-D array shown in Figure 5.3 contains partial reconfiguration tiles sized at 9,600 LUTS, 360KB BRAM, and 80 DSPs. This particular configuration was sized to hold the largest bitstream generated from one of our DSL test suites. Setting the size of the tiles occurs when the DSL is

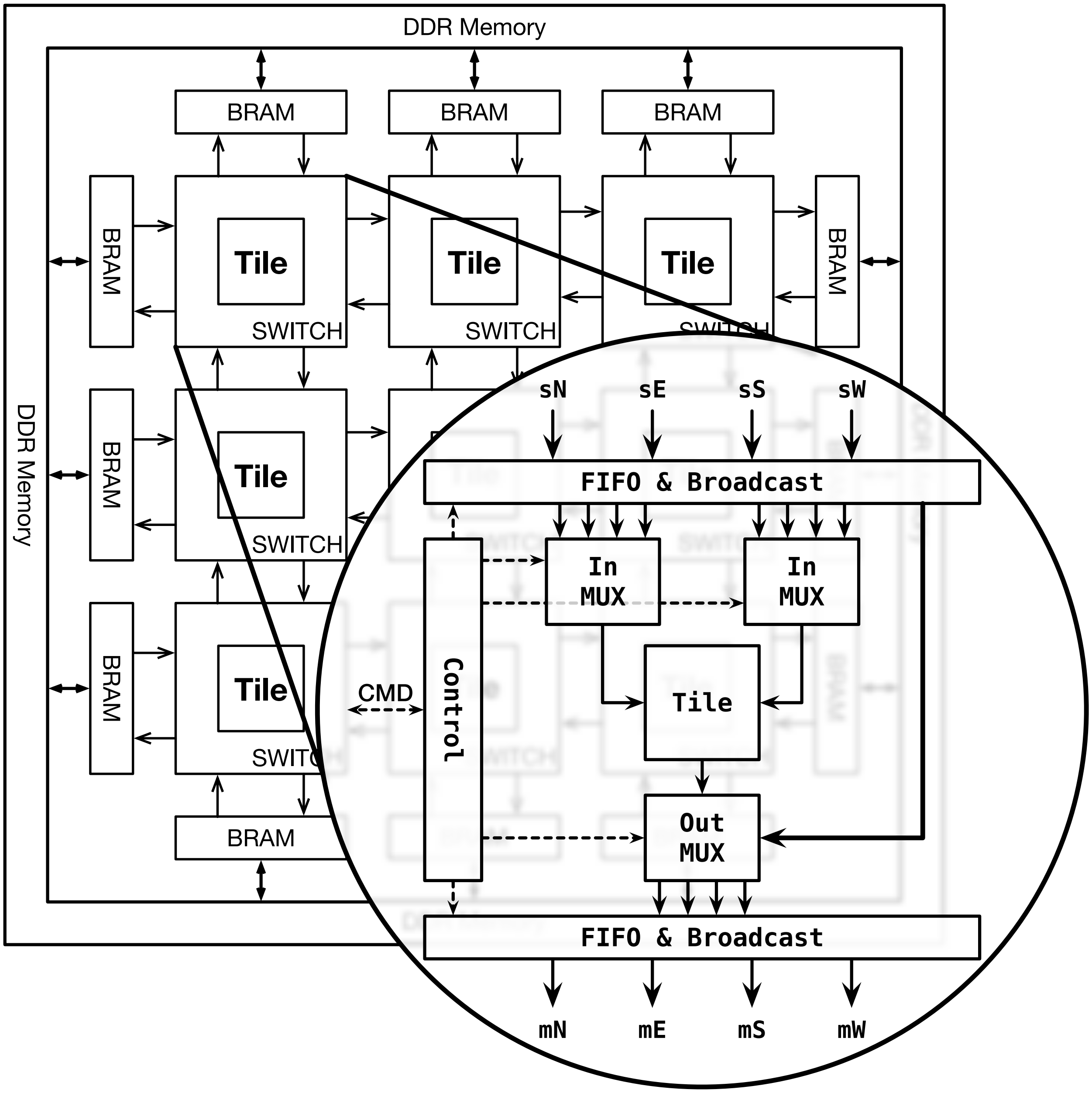
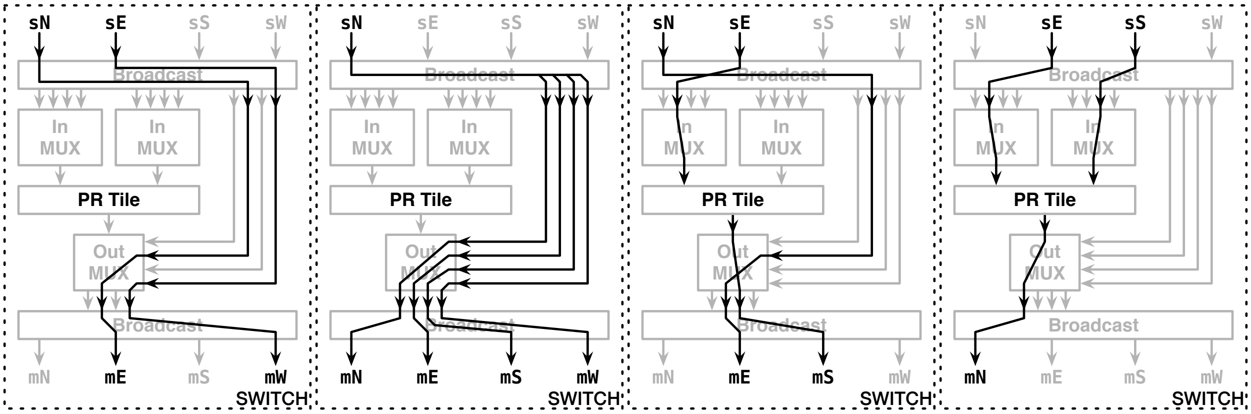


Figure 4.3: 3 × 3 Tile Array and Interconnect Network.

first created. The number of the tiles is derived based on the size of the tile and target FPGA logic family. We have automated the creation of the overlay; switches, buffers, interconnects BRAMs, and tiles for Xilinx FPGAs. Our automation tool produces a TCL script that can be input into Xilinx's Vivado tools. The current floorplanning tool requires that that logic for each programming pattern be placed in each tile to generate bitstreams. Although not efficient, creating the overlay only occurs once per board and is not in the path of the application programmer. Thus this inefficiency does not negate the approaches ability to allow programmers to JIT accelerators at run time.

4.2.2 Programmable Switch

Figure 4.4 provides an exploded view of a switch. Figure 4.4 shows the types of routing patterns that can be programmed into the switch. Routing patterns were defined to enable each switch to direct any input into, as well as output from the tile. Switches support pass through connections for routing between distant tiles. Figure 4.4(a) shows how two inputs can be configured to pass data to two different outputs. Figure 4.4(b) shows the switches broadcast capability, which allows a single input to fan out data on multiple output routes. Figure 4.4(c) shows the switch supporting a unary streaming model, with one input being fed into the tile, and results fed back out. Figure 4.4(d) shows a typical two input, one output pass through the tile. Routes can be set statically or dynamically. Dynamic settings can be used for allowing the switch to support different time varying routing needs such as when multiple accelerators are resident within the overlay. Each switch may serve as a pass through for one accelerator, and then source and synch data for a tile that is part of a different accelerator. The switch logic and interconnects are currently implemented (on Xilinx FPGAs) using standard 32 bit AXI streaming interconnects.



(a) (b) (c) (d)

Figure 4.4: Switch Routing.

4.2.3 Local Memory

The boundary cells in the overlay include connections to blocks of local memories (BRAMs). These BRAMS can be used as addressable local memories or as FIFO data buffers for streaming data. The size and number of BRAMs are variable and can be configured when the overlay is created. Block data transfers use DMA (not shown) between the BRAMs and Global DRAM memory. The BRAMs are placed within the global address map of the system, allowing any processor or bus master device to transfer data into and out of a local memory. The BRAMS have buffer full/empty handshaking signals that are connected through the switches to enable processing to be dynamically triggered.

4.3 Interpreter

Table 4.1 lists the set of platform independent interpreter calls produced by the backend of our compiler. This approach allows the same accelerator to be JIT assembled on different configurations of overlays. This is shown in Figure 4.1 where the same interpreter calls are executed on three different configurations of our overlay embedded within different FPGAs. The simple example in Figure 4.1 shows the relationship between the data flow graph information extracted from our compilers intermediate representation (LMS in our Delite compiler framework) and the interpreter calls. In this simple example the Delite compiler first generates two Intermediate Representations (IR) for the DSL programming patterns: map (Map) followed by reduce (REDU). Then, it generates two interpreter call as follows:

- VAM_GET_TILE (&VT, &PR1)
- VAM_GET_TILE (&VT, &PR2)

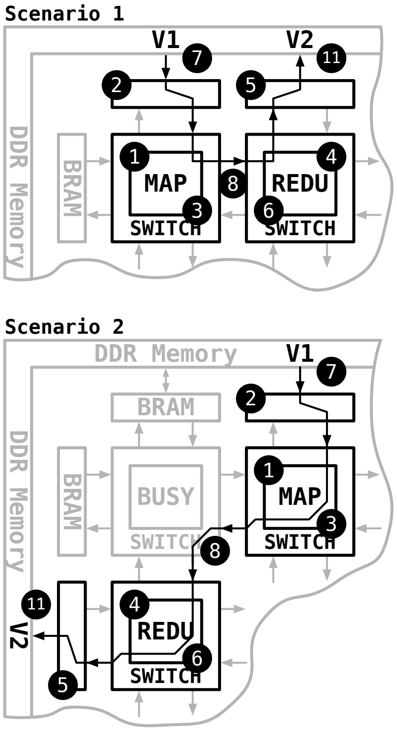
The VAM_GET_TILE assigns bitstreams to tiles within the local overlay. PR1 and PR2 are pointers to the bitstreams for map and reduce that reside in memory respectively. These calls place no impositions on how an interpreter manages or allocates its machine specific resources. Management of tiles is performed by each interpreter. The call simply directs the interpreter to find and transfer the bitstreams into the available resources (a partial reconfiguration tile in our overlay). This allows the interpreters running on the three systems shown in Figure 4.1 to manage and allocate resources differently. The interpreters on each of these three systems transfer the bitstreams into their selected tiles using:

- VAM_LOAD_TILE (&BS_T, PR1, MAP)
- VAM_LOAD_TILE (&BS_T, PR2, REDU)

The locations of the tiles that were loaded with the bitstreams are returned to the interpreter through the &PR1 and &PR2 variables. These variables are then used to form the data paths between the bitstreams using the following interpreter command:

- VAM_ROUTE (&VT, PR1, PR2, 2)

The interpreter uses the PR1 and PR2 variables to set the interconnects between the tiles that hold the bitstreams. This interpreter call does not bind specific mappings of the data flow graph to a any predetermined configuration of connection and switch boxes, or channels. The specific configuration of the intermediate fabric is only known to the interpreter running on each platform. This separation of policy and mechanism allows the interpreter to assemble, place and



DSL: V2 = V1.Map(e×e).Reduce((a,b)= a+b)

```
1 VAM_GET_TILE(&VT,&PR1);
2 VAM_GET_BRAM(&VT,PR1,BRAM,1,L1);
3 VAM_LOAD_TILE(&BS_T,PR1,MAP);
4 VAM_GET_TILE(&VT,&PR2);
5 VAM_GET_BRAM(&VT,PR2,BRAM,1,L2);
6 VAM_LOAD_TILE(&BS_T,PR2,REDU);
7 VAM_DMA(&vdma,V1,BRAM,L1);
8 VAM_ROUTE(&VT,PR1,PR2,2);
9 VAM_START(PR1,PR2,2,L1,L2);
10 VAM_DONE(&VT,PR1,PR2,2);
11 VAM_DMA(&vdma,BRAM,V2,L2);
```

Figure 4.5: VAM Call Flow With Different Cases.

route the accelerator at run time. The same separation of policy and mechanism bring portability over different intermediate fabrics.

4.4 Interpreter Functionality

In this section we continue with our running example shown in Figure 4.1 to illustrate how the interpreter executes the calls on the two overlay configurations shown in Figure 5.4.

Function Placement and Loading: We chose to manage the free PR tiles in a simple queue (the VAM_TABLE). For each ***VAM_GET_TILE*** (steps 1, 3) the interpreter pops a free tile from the queue. The tiles returned for two consecutive ***VAM_GET_TILE*** calls may not be adjacent within the overlay array. This is shown in Figure 5.4. On the left the interpreter selected two adjacent tiles while on the right, the top right and bottom left tiles were selected. Function bitstreams are then loaded into free tiles using ***VAM_LOAD_TILE*** (steps 3, 6). The run time interpreter performs this operation by DMA'ing the bitstream resident in DRAM into the ICAP port of our (Xilinx) FPGA. The run time interpreter manages input and output buffers for the accelerator in a similar fashion to tiles. For each input variable the ***VAM_GET_BRAM*** (steps 2, 5) returns a list of available local BRAMs to be selected as an input buffer. The BRAM buffers do not have to be adjacent to the tile that is holding the bitstreams. This is shown on the right in Figure 5.4 where the input BRAM connected to the top left tile along with the BRAM connected to the top right tile were returned as available. The switch boxes will be set to pass through mode as part of the VAM_GET_BRAM call to allow data transfers between remote BRAMS and tiles.

Function Routing: After the interpreter transfers the bitstreams into the tiles, and BRAMs are selected, data paths are formed from the ***VAM_ROUTE*** (step 8) calls. For our prototype systems we implement a simplified version of the standard maze-routing algorithm [14]. The right side of Figure 5.4 shows the route formed from the top right tile (MAP) to the bottom left tile (REDU). This route first traverses down to the top left tile, and then down to the bottom left tile. Once the maze routing algorithm is run, the switch boxes are set between the tiles. Note that the top right tile on the left side of Figure 5.4 is set in a pass through mode, routing the east input to the south output. The top left tile can still be used for hosting different bitstreams while the switch configures the pass through mode for this example. The simple routing algorithm as well as the routing capabilities of the switches allow multiple accelerators to be hosted within this 3×3 tile array overlay.

Data Transfer: After the accelerator has been configured, the interpreter transfers input data from DRAM into the local input buffer BRAMS using ***VAM_DMA*** (steps 7). The outputs of the accelerator are transferred from the output BRAM buffer back into DRAM using ***VAM_DMA*** (step 11).

Control Operations: The ***VAM_START*** (step 9) initiates the execution of the array. The ***VAM_DONE*** (step 10) returns status from the accelerator. These two steps are not presented on the left part of Figure 5.4.

Patterns	Expression	BRAM	DSP48E	FF	LUTs
Map	*Map(e×e)*	0	3	410	793
Map	*Map(sqrtf(e))*	0	0	643	1,487
Map	*Map(log(e))*	0	61	2,240	1,894
Map	*Map(if(e>0) +1 else −1)*	0	0	169	208
Reduce	*Reduce((a,b)=a+b)*	0	2	638	761
Reduce	*Reduce((a,b)=if(a<b) a+b else a−b)*	0	0	211	309
Filter	*Filter(e>x)*	0	0	170	209
Filter	*Filter(e<x)*	0	0	170	209
Filter	*Filter(e==x)*	0	0	170	209

Table 4.2: Pre-Synthesized Parallel Pattern Samples

4.5 Experimental Results and Analysis

Figure 4.1 shows three unique systems created to evaluate our approach. The first system contained a 2×2 array built on a Kintex 7. The second contained a 3×3 overlay on a Virtex 7. The third system was built on a Virtex 7 and contained two 2×2 overlays. In all cases the overlays were interfaced to a MicroBlaze processor as tightly coupled accelerators. Figure 4.1 only shows MicroBlazes and overlays for clarity sake. Each system additionally included BRAMs to host the operating system and interpreter, busses, I/O and support devices, DMA and ICAP devices, and global DRAM. The overlays were created through our automated script and produced TCL scripts that were input into the CAD tools for synthesis. All systems were built and synthesized using Vivado 14.2 tools.

System software included our pthreads compliant middleware and operating system hthreads that enabled multithreading on the MicroBlazes. The interpreter was written in C and cross compiled with the operating system. As all systems used a MicroBlaze we were able to compile the interpreter once and reuse it on all systems. Interpreter calls are invoked through executing sys_calls. Sequential portions of the test programs were cross compiled and run as a thread, or in the case of mulitprocessor system, as concurrent threads on the two MicroBlazes.

4.5.1 Creating the Accelerators

The first column in Table 4.2 list part of the programming patterns adopted from OptiML (an existing DSL within Delite) [7]. These are familiar programming patterns from machine learning and high performance computing applications. The second column lists the expressions that can be executed for each programming pattern. The $sqrtf(e)$ and $log(e)$ operate on single precision floating point operators. All remaining functions operate on integers. Prototypes (function definitions) were created for each programming pattern, and the expressions within each programming pattern were coded in C as part of the DSL creation process. The C bodies were passed through Vivado HLS to generate bitstreams. We added an additional flag to the standard Delite compilation flow to allow the C versions of the DSL to be compiled for test and evaluation, or cross compiled to run on the MicroBlaze processors for comparison. Switching the compiler flag was all that was needed to generate interpreter calls with symbolic links to the bitstreams. The remaining three columns show the resources used to implement each expression.

We used the patterns and expressions listed in Table 4.2 to create the four benchmark accelerators listed in Table 4.3. These benchmarks are illustrative representations of how a programmer would compose the *Map*, *Reduce*, and *Filter* patterns into an accelerator. The composed expressions were compiled using the Delite compiler front end. Our VAM call generator backend produced interpreter instructions. It is important to note that these expressions could be changed, and new expressions created by compiling and without synthesis.

The run time interpreters running on each system shown in Figure 4.1 executed the interpreter calls, including transferring the bitstreams for each expression and setting the interconnects between the tiles and sequencing the accelerator. In summary each benchmark was compiled once and the output run on the multiple platforms. This verified the portability of the interpreter calls over different versions of our overlay. Importantly each benchmark was created by compiling the composed pattern expressions without having to synthesize. This is a fundamental step in getting application developers to use FPGAs; CAD tools and synthesize need

Composed Patterns	Approach	Total (μs)	Overhead (μs)	DMA (μs)	Acc (μs)	Speed up
V2 = V1.Map(e×e) .Reduce((a,b)=a+b)	Software	1,611	-	-	-	1×
	HW (Full Module)	309	-	63	246	5.2×
	HW (JIT)	348	39	63	246	4.6×
V2 = V1.Filter(e>x) .Map(e+const.)	Software	2,021	-	-	-	1×
	HW (Full Module)	186	-	104	82	10.9×
	HW(JIT)	225	39	104	82	9×
V2 = V1.Filter(e>x) .Map(log(e))	Software	1.4*s*	-	-	-	1×
	HW (Full Module)	882	-	104	778	1,579×
	HW(JIT)	921	39	104	778	1,512×
V2 = V1.Filter(e>x). Map(log(e)) .Reduce((a,b) = if a>b, a−b else, a+b)	Software	1.4*s*	-	-	-	1×
	HW (Full Module)	1,046	-	63	983	1,331×
	HW(JIT)	840	39	63	738	1,659×

Table 4.3: Performance

to be removed from their development paths.

4.5.2 Discussion: Performance Analysis

It was anticipated that run time assembling accelerators would suffer some measure of degraded performance compared to a single custom synthesized version. We further anticipated that our initial prototypes would suffer additional performance degradations compared to later optimized revisions. Clearly the performance of any accelerator is dependent on many different factors, including how the code is structured, the time taken to optimize the code, and the designers hardware design skills. We made every attempt to apply the same types of coding style to the creation of both custom accelerators and programming patterns to eliminate any bias in comparing performance. To set a base case for comparison we also ran a software version of each benchmark on the MicroBlaze. We used the execution time of the software to compute relative and fair speedups for the synthesized version and the assembled at run time using our JIT approach. The run time results shown in Table 4.3 were generated on a Virtex 7 on the 3×3 overlay. Thus we focus our discussion on the execution times reported in Table 4.3.

We were intrigued to observe the speedups were approximately equivalent to a synthesized custom accelerator. The slight difference in speedup is attributed to the overhead of setting up the tile array. This overhead would be incurred once, when the accelerator is first assembled. This overhead would not be seen when the accelerator is invoked a second time. While the results are promising we are reluctant to draw any conclusions on performance based on these relatively few and simple benchmarks. From a conservative perspective what we conclude is that the results simply do not negate the validity of the approach. Clearly, more DSLs and more applications need to be evaluated before any meaningful performance trends can be reported. What can be inferred is that the approach does allow a programmer to rapidly create and evaluate the execution times of accelerators. At a minimum the approach represents a powerful capability for rapidly prototyping and evaluating the performance of accelerators.

The interpreter was implemented in software as part of the operating system running on a MicroBlaze. In our preliminary work we did run test applications to verify the ability to run time assemble different accelerators. The performance relationship between run time assembling and a custom accelerator is identical to the results shown in Table 4.3.

4.6 Conclusion

A new approach was presented to enable programmers to use standard software development flows to create hardware accelerators and bypass CAD tools and synthesis. The approach introduced a new PR tile overlay and set of interpreter calls that brings portability into the process. This will greatly facilitate the use of FPGAs within our software dominated information technology sector. Results were presented showing a complete end to end capability; from working within a DSL to assembling the accelerator at run time. Results also show the costs in terms of additional resource overheads for the accelerator functionality as well as the overlay.

Future Work

Building the prototype raised just as many questions as it answered. Determining the geometry and interconnect of PR tiles needs a more quantitative treatment. Compiler optimizations were turned off when creating the bitstreams for each programming pattern. Further investigations are needed to determine what types of optimizations should be applied to each programming pattern before they are synthesized. Our next major goal is continue our investigation on a large at-scale data center computer with FPGAs running tuned DSLs. Such at scale systems are beginning to become available for research. Running real applications and data loads on at scale systems will allow us to better evaluate performance and area costs for refining the approach. This will also allow us to transition the approach into the hands of the programmers for further study.

References

[1] CRA. Revitalizing Computer Architecture Research.

[2] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Jim Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *41st Annual International Symposium on Computer Architecture (ISCA)*, June 2014.

[3] Timothy Morgan. Intel Mates FPGA With Future Xeon Server.

[4] Hyouk Joong Lee, K.J. Brown, A.K. Sujeeth, H. Chafi, K. Olukotun, T. Rompf, and M. Odersky. Implementing domain-specific languages for heterogeneous parallel computing. *Micro, IEEE*, 31(5):42–53, Sept 2011.

[5] N. George, Hyoukjoong Lee, D. Novo, T. Rompf, K.J. Brown, A.K. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne. Hardware system synthesis from domain-specific languages. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8, Sept 2014.

[6] David Andrews, Ron Sass, Erik Anderson, Jason Agron, Wesley Peck, Jim Stevens, Fabrice Baijot, and Ed Komp. Achieving Programming Model Abstractions For Reconfigurable Computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(1):34–44, January 2008.

[7] A.K. Sujeeth, H. Lee, K.J. Brown, T. Rompf, H. Chafi, M. Wu, AR Atreya, M. Odersky, and K. Olukotun. Optiml: An implicitly parallel domain-specific language for machine learning. In *Proceedings of the International Conference on Machine Learning. Haifa, Israel*, 2011.

[8] 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*, pages 13–22, Oct 2010.

[9] 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*, pages 21–24, May 2015.

[10] D. Capalija and T.S. 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*, pages 1–8, Sept 2013.

[11] Jason Yu, Guy Lemieux, and Christpher Eagleston. Vector Processing as a Soft-Core CPU Accelerator. In *FPGA '08: Proceedings of the 16th international ACM/SIGDA Symposium on Field Programmable Gate Arrays*, pages 222–232, New York, NY, USA, 2008. ACM.

[12] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. VESPA: Portable, Scalable, and Flexible FPGA-Based Vector Processors. In *CASES '08: Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 61–70, New York, NY, USA, 2008. ACM.

[13] E. Cartwright, A. Fahkari, S. Ma, C. Smith, M. Huang, D. Andrews, and J. Agron. Automating the design of mlut mpsopc fpgas in the cloud. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 231 –236, aug. 2012.

[14] C.Y. Lee. An algorithm for path connections and its applications. *Electronic Computers, IRE Transactions on*, EC-10(3):346–365, Sept 1961.

Chapter 5

Breeze Computing: A Just In Time Approach for Virtualizing FPGAs in the Cloud

Sen Ma, David Andrews, Shanyuan Gao, and Jaime Cummins

In this paper, we introduce a new design flow and architecture that lets programmers replace synthesis with compilation to create custom accelerators within data center and warehouse scale computers that include reconfigurable manycore architectures. Within our new approach, we virtualize FPGAs into pre-defined partially reconfigurable tiles. We then define a run time interpreter that assembles bitstream versions of programming patterns into the tiles. The bitstreams as well as software executables are maintained within libraries accessed by the application programmers. In our approach, synthesis occurs hand in hand with the initial coding of the software programming patterns when a Domain Specific Language is first created for the application programmers. Initial results show the approach allows hardware accelerators to be compiled 100× faster compared to the time required to synthesize the same functionality. Initial performance results further show a compilation/interpretation approach can achieve approximately equivalent performance for matrix operations and filtering compared to synthesizing a custom accelerator.

5.1 Introduction

Interest has been growing in using FPGAs within internet based cloud computing and data center systems. This interest has been piqued with two recent announcements. First Microsoft announced Catapult which integrated FPGAs into a data center to accelerate their Bing search engine. Analysis verified the energy efficiency of mapping key portions of their document ranking algorithm into the FPGAs. Performance was doubled at only a 30% increase in energy [1]. Secondly Intel announced the acquisition of Altera. This was followed by Intel's announcement to integrate FPGAs with large Xeon multi-core processors [2]. Making commercially available reconfigurable manycore architectures is part of Intel's strategy to increase their market share for large data analytics applications running in data centers and warehouse scale computers.

Successfully transitioning FPGAs into commercial data centers will require integrating FPGAs under the software centric virtualization ecosystems populating commercial data centers [3]. Among the many issues that challenge such a successful integration is how to enable data center programmers to create accelerators within FPGAs [4]. This goal has yet to be fully met, but historical research targeting single Systems on Chip architectures from the embedded systems domain provides important directions.

Prior research in programming languages has led to the commercial availability of High Level Synthesis (HLS) synthesis tools. These tools allow gates to be efficiently synthesized from subsets of C code. HLS is not a Panacea for programmers in the data center. HLS still requires knowledge of hardware design, the use of vendor specific CAD tools, and time consuming synthesis, place and route.

George et. al. [5] showed how detailed knowledge of hardware design could be moved out of the application designers path when using Domain Specific Languages (DSLs). George argued that HLS generation and hardware centric modifications should happen once when the functionality of a programming pattern was implemented. Programmers could then create accelerator functionality by combining and then synthesizing the pre-coded, optimized programming patterns. Similarly, Xilinx recently released their SDSoC software framework that incorporates HLS under a broader eclipse framework for their Zynq family of devices. The SDSoC tool automates the generation of the accelerators including Hw/Sw interfaces and synchronization support from a single threaded source code. These approaches are important steps to remove the need for programmers to understand hardware design. However they still require circuits to be synthesized within hardware centric CAD tools.

Research has also investigated how libraries of pre-synthesized accelerators can be leveraged to bypass synthesis. Libraries of accelerators created under partial reconfiguration rules allow different accelerators to be swapped in and out at run time. Library approaches by themselves do not offer a path for programmers to modify or create new accelerators without having to pass through synthesis.

Yet another interesting approach is to create intermediate architectures, or overlays. Overlays raise the level of abstraction of what is programmed within the FPGA from low level LUTS and FF's to components than can be targeted by a compiler. Overlay components range from Arithmetic Logic Units, Vector Accelerators [6] and even Multiprocessor Systems on Programmable Chips (MPSoPCs). Overlays tradeoff the ability to customize the lower level circuits for each application for the ability to compile to a stable set of op_codes or ISA.

Each of these efforts address key aspects of the problem, but by themselves each fall short of providing a complete end to end solution. Creating such a complete end to end solution is the driving goal of our research. In fact the stretch goal of our research is to completely abstract away any differences for data center programmers when creating new functionality to be implemented as circuits in a hardware accelerator or software executables.

To meet our driving goal we have created a new design flow that fits within the existing virtualization ecosystems in use in data centers. Our approach incorporates and builds upon key aspects of the research efforts discussed above. Similar to George et. al. [5] we advocate the use of Domain Specific Languages (DSLs) to allow data center programmers to create high level descriptions of accelerator functionality using pre-coded programming patterns. Our approach takes the additional step of then removing the need to synthesize each new accelerator design by providing a library of pre-synthesized programming patterns. This allows data center programmers to create new functionality by compiling links to the bitstream versions of the programming patterns within a compiler. We retarget the output of the compiler to generate a data flow graph that captures the control and data flow dependences into and out of the accelerator, as well as between the programming patterns within the accelerator.

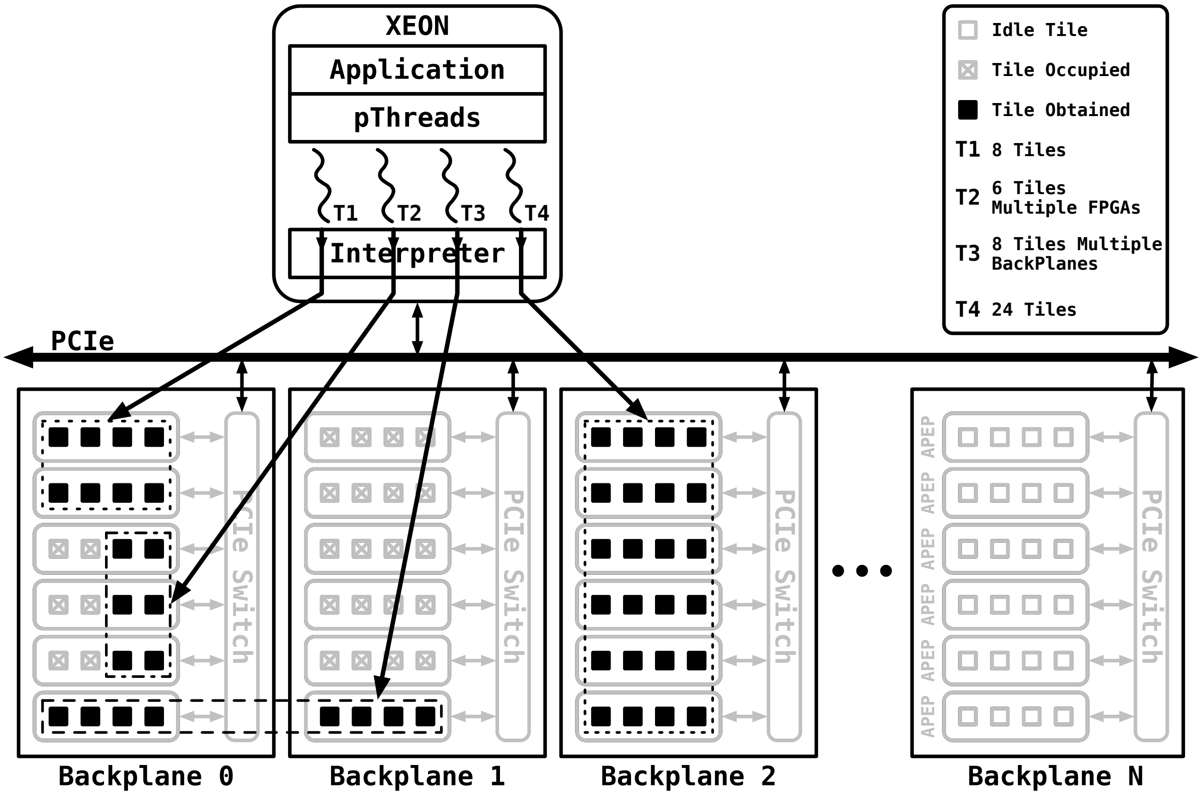


Figure 5.1: Multiple Tasks Mapping on Cloud System Through Run Time Interpreter.

Figure 5.1 shows our approach to modifying the run time system and the creation of a new overlay architecture. We provide a programmable overlay that replaces programmable ALUs, or soft core processors with tiles of free gates that can be customized under partial reconfiguration rules. This extends the concept presented in [4] that targeted single chip SoC systems. A restriction of this architecture was the requirement that a single accelerator fit within a single FPGA. We introduce a new abstraction layer and interconnect network that allows single accelerators to span over multiple FPGAs. All FPGAs are managed as a common virtual sea of reconfigurable gates more appropriate for data centers and reconfigurable cloud systems that will contain multiple FPGAs shared between multiple threads, tasks, and programs. Figure 5.1 shows how our approach enables multiple threads running on a Xeon to spawn different accelerators, each of which uses tiles that can span across multiple FPGAs.

5.1.1 Contributions

The main contributions of this work are:

- *Dynamically Reconfigurable Virtual Overlay* A new virtual overlay using partial reconfiguration (PR) tiles that abstracts accelerators over multiple FPGA resources. The overlay represents a framework within which the run time interpreter assembles accelerators.
- *Automated Overlay Generator* We provide a scripting tool that automatically creates different overlay configurations based on parameterized inputs.
- *Run Time Interpreter* We created a reentrant interpreter that runs on a Xeon to manage all FPGA resources, including allocating or releasing PR tiles, transferring data within or between virtual FPGAs, and configuring the PR tiles. The run time interpreter is written in C and is portable over different host processor families.
- *Platform Independent Interpreter Language* A prototype set of platform independent interpreter calls that enable portability of code over different configurations of overlays. This represents a separation of policy from mechanism that allows a single set of interpreter calls to be implemented by all platform specific interpreters managing different overlays.
- *Standard Accelerator Interface Template* We establish a set of standard hardware interfaces for hardware designers to use when creating new bitstreams to integrate into the standard communications interface of our overlay.

The work reported in this paper has been implemented and evaluated on a reconfigurable cloud computer that contains two Xeon-2650 manycores, each with 8 cores connected to 23 user accessible Xilinx Kintex7 FPGAs. The run time system is based on pthreads and Linux.

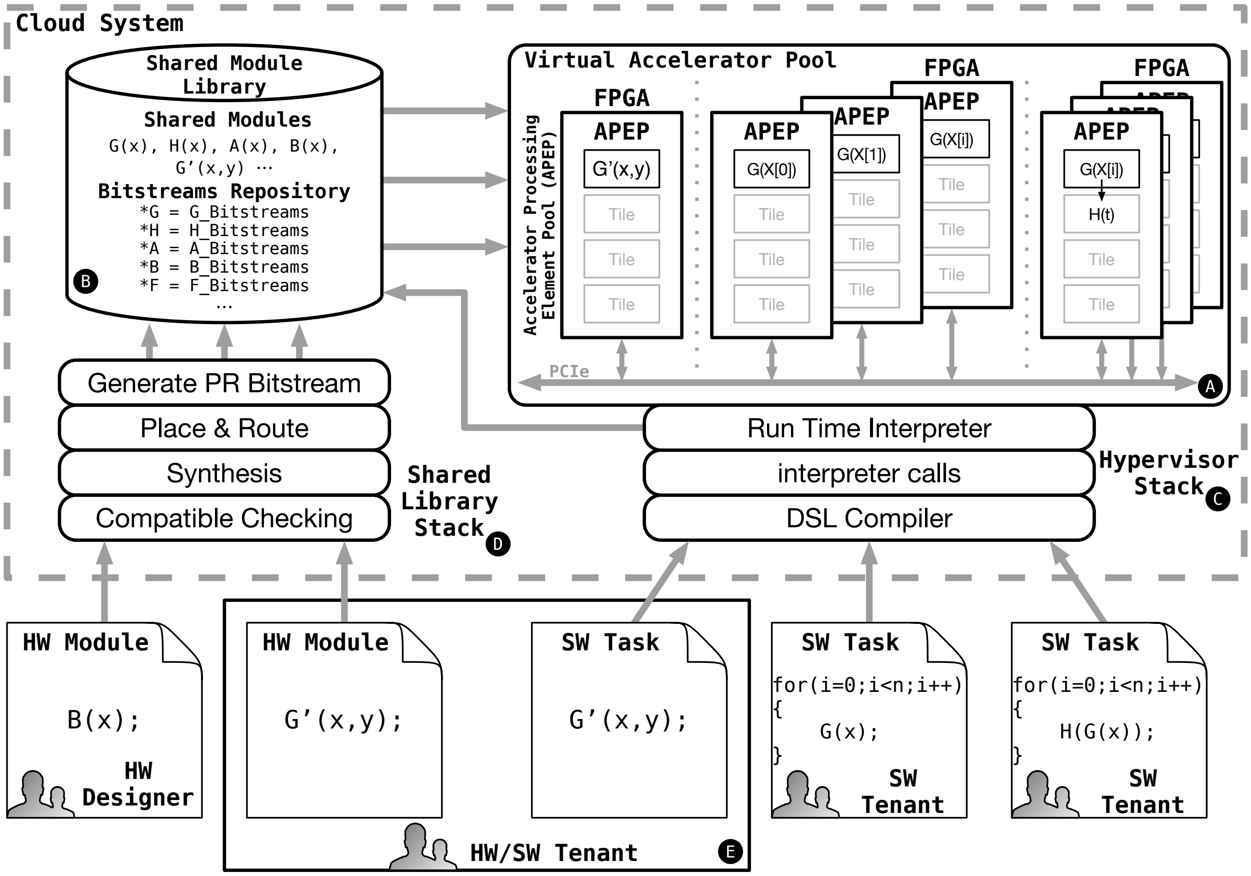


Figure 5.2: Design Flow.

5.2 User Front End

Figure 6.6 shows a high level description of our compilation flow and architecture. This flow has been developed to support three types of users.

SW Tenets The bottom right shows how data center programmers, or Tenets, use our flow to compile new accelerators. SW Tenets form accelerator functionality using the programming patterns archived in the library (Module B top left). Programmers do not work with the bitstreams but instead are provided software function prototypes of the programming patterns to include in their code. This allows data center programmers to remain within familiar compiler frameworks for creating new functionality. The software function prototypes are compiled and linked just as if they were prototypes to dynamically linked software libraries. Programmers have access to both software executables as well as bitstream versions of the programming patterns. This allows programmers to first create and verify the target functionality for the accelerator in software. After the functionality has been verified a flag within the compiler is set that directs the compiler to switch the link from the software executable to the hardware bitstream.

We provide a simple example to show the flexibility of the approach. Consider how the two separate hardware accelerators $acc_1 := H(G(x));$ and $acc_2 := G(H(x));$ can be created. Under current approaches the HLS representation of H(x) and G(x) for *acc*_1 would need to be combined first and then synthesized. Even though *acc*_2 uses the same G(x) and H(x) programing patterns the fact that they are combined in a different order requires resynthesis. Our approach eliminates the need to synthesize either accelerator. Instead data center programmers can compose the function prototypes for *G*(X) and *H*(X) in any organization without having to pass through synthesis. From the data center programmers perspective there is no different between compiling and linking to software or hardware executables.

We achieve this transparency between generating software and hardware through a standard modification to the backend of the open source Delite compiler framework [7] used in this project. In place of generating a specific processor ISA, for hardware accelerators we generate interpreter calls with links to the bitstreams. The interpreter calls are presented in section 5.4. Delite has an Intermediate Representation (Lightweight Modular Staging or LMS) that forms a data flow graph description of how the function prototypes were composed. The modular structure of the Delite frameworks allowed us to add in our new backend code generator along with the currently existing suite that comes with the framework. Thus generating our interpreter calls was a straight forward engineering exercise and will not be discussed further.

Hw Designers The bottom left of Figure 6.6 shows the path used by hardware designers. Synthesis still must occur, however we move this path out of the data center programmer's design flow. Hardware designers create and enter new bitstream versions of programming patterns into the library that will then be shared and reused by the data center programmers. We provide a standard interface template for use by all hardware designers. This interface along with several utility scripts can be used to allow the hardware designer to create PR bitstreams that are compatible with the standard communication interface of our overlays. We have developed a script for Xilinx based FPGAs that automates the creation of bitstreams for systems such as our overlay with multiple PR regions. This allows the hardware designer to specify the programming pattern for a standard PR tile once, and the script then places and synthesizes the programming pattern within all PR tiles. After the programming pattern is synthesized it can be checked into the shared library for use by all data center programmers.

Hw/Sw Tenets The bottom center shows how designers with both software and hardware experience can create new hw/sw co-designed applications. This path would be used for traditional users to accelerate their design with custom accelerators not included in the library.

5.3 Virtualization Architecture

Part A (top right) in Figure 6.6 shows the creation of a vendor neutral overlay called the accelerator processing element pool (APEP). We provide an automated script that can be used to generate an APEP. The script requires the designer to set few inputs needed to specify the target APEP geometry. Our current scripts output all files needed to synthesize the APEP using the Vivado CAD tools. It is important to note that the generation of the APEP occurs once during the initial design of the overlay. Once designed, the APEP is placed in the shared library to be used by data center programmers as well as hardware designers implementing the bitstream versions of programming patterns. Part A also shows the (static) APEP overlay being replicated within multiple FPGAs to compose a larger virtual accelerator pool (VAP).

The interconnect structure of the APEP is flexible and can be tailored to support the bandwidth and processing requirements of the programming patterns for a particular DSL. Each

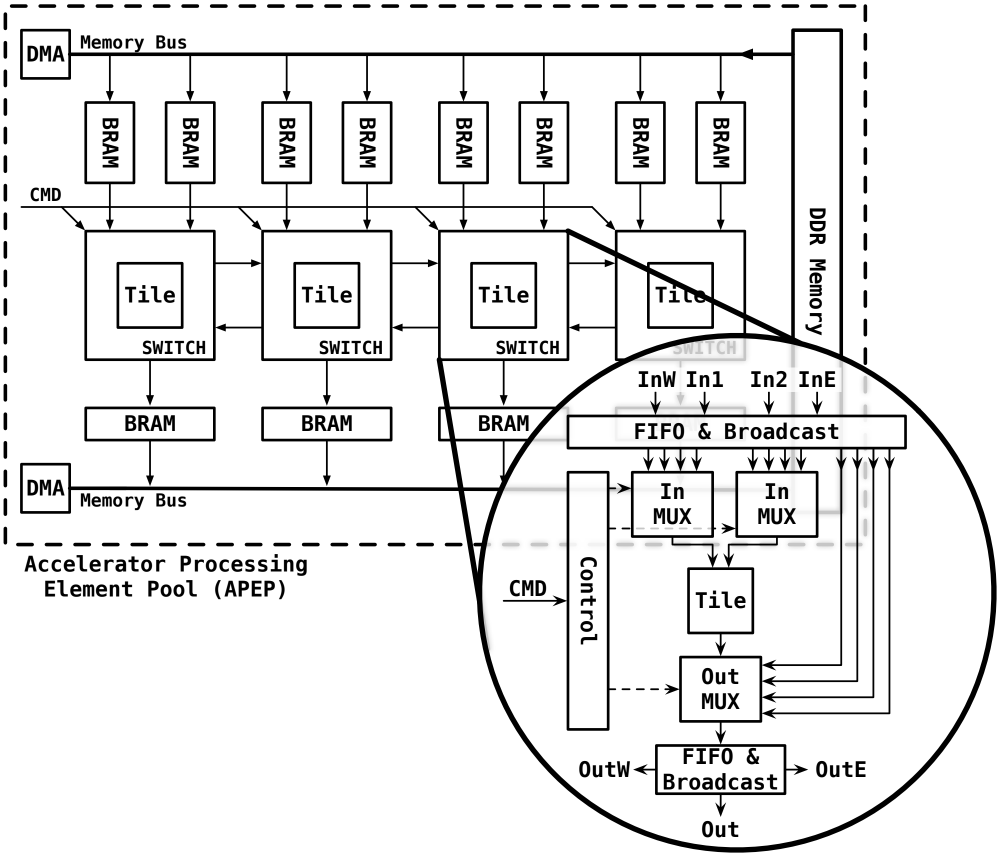
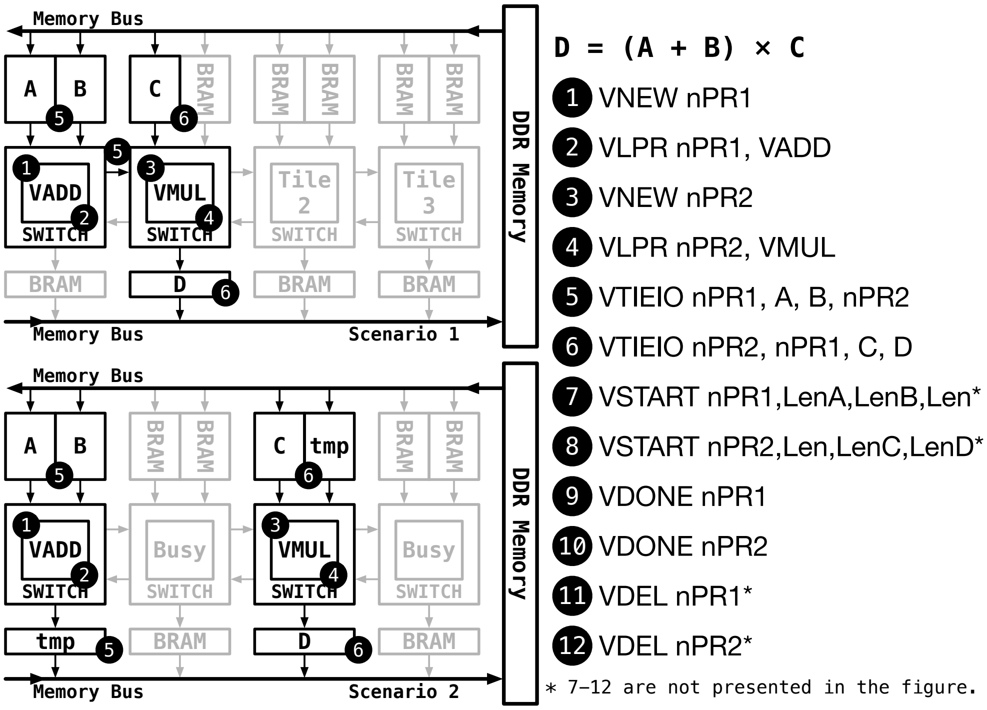


Figure 5.3: Architecture of accelerator processing element pool (APEP).

APEP can be transparently replicated across multiple FPGAs. The run time interpreter discussed in section 5.4 manages all APEPs as a virtual accelerator pool. Figure 5.3 is an example of a simple APEP architecture created for this paper. This APEP is populated with a 1×4 array of partial reconfiguration tiles, with each tile containing 15,600 LUTS, 60 RAMB18, and 60 DSPs. This particular configuration was sized to hold the largest bitstream generated for the programming patterns used to form the benchmarks presented in this paper. When replicated across the 23 FPGAs in our experimental system, the system represents a virtual accelerator pool with 92 PR tiles available to assemble the programming patterns from the software threads running on the Xeon.



D = (A + B) × C

1 VNEW nPR1
2 VLPR nPR1, VADD
3 VNEW nPR2
4 VLPR nPR2, VMUL
5 VTIEIO nPR1, A, B, nPR2
6 VTIEIO nPR2, nPR1, C, D
7 VSTART nPR1,LenA,LenB,Len*
8 VSTART nPR2,Len,LenC,LenD*
9 VDONE nPR1
10 VDONE nPR2
11 VDEL nPR1*
12 VDEL nPR2*

* 7-12 are not presented in the figure.

Figure 5.4: VAM Call Flow With Different Cases.

5.3.1 Local Memory

As shown in Figure 5.3 tiles include connections to blocks of local memories (BRAMs). These BRAMS can be used as addressable local memories or as FIFO data buffers for streaming data between tiles, and between tiles and DRAM. The size and number of BRAMs are variable and are configurable when the overlay is created. Block data transfers either use DMA between the BRAMs and Global DRAM memory or directly use a streaming protocol. The BRAMs are placed within the global address map of the system, allowing any processor or bus master device to transfer data into and out of a local memory. The BRAMS have buffer full/empty handshaking signals that are connected through the switches to enable flow control between producers and consumers.

5.4 Run Time Interpreter

When modifying the backend of the compiler to generate descriptions of the control and dataflow requirements of the accelerator, processor specific instructions could have been output from the compiler. However generating ISA specific code would have required re-targeting the compiler to generate code for every platform and variance of tile-communication overlay. This also would have required knowledge of specific routing algorithms used for each platform. Our approach was to target the compiler to output architecture neutral interpreter calls that could then be implemented by interpreters running on each platform, each knowing their own specific platform and tile-communication overlay. This approach follows the JIT approach used in software to bring portability into the mixed architectures found within data centers.

To support this approach, we created a set of platform neutral interpreter calls that represented the standard operations that the run time system would need to perform for assembling and executing accelerators within our overlay. We baselined these calls and wrote the functionality of the interpreter in C. The C source of the interpreter can be easily modified to run on any processor and control different configurations of the APEP overlay. Figure 5.4 shows a subset of the baselined interpreter calls that are generated from the back-end of our compiler. Baselining the calls allows the data flow graph representation of an accelerator to be interpreted within different platform specific configurations of tiles and overlays. We have successfully run a single set of interpreter calls on stand alone MPSoPC systems as well as our prototype reconfigurable cloud. Figure 5.4 shows a simple example of how interpreter calls can be implemented in two different scenarios by the run time interpreter. The next sections discuss the semantics of the calls.

Function Placement and Loading: The interpreter uses the ***VNEW*** (Step 1, 3) calls shown in Figure 5.4 to obtain free PR tiles within an APEP. The interpreter returns *nPR1* and *nPR2* which are pointers to the free tiles obtained for the vector add (VADD) and vector multiply

(VMUL) programming patterns respectively. The allocated tiles need not be adjacent, or even within the same FPGA. This is a decision that is up to the interpreter that manages all tiles regardless of chip location as a pool of available resources. Scenario 1 shows the interpreter returning two adjacent tiles, and scenario 2 shows the return of non-adjacent tiles. Regardless of location the bitstreams for the programming patterns are then loaded into free tiles using ***VLPR*** (steps 2, 4). The run time interpreter DMA's the bitstreams from DRAM into the ICAP port of our (Xilinx) FPGA to program the tiles as part of this call.

Function Routing: For each input variable, the ***VTIEIO*** (steps 5, 6) directs the interpreter to configure data paths between tiles, or between tiles and buffers. The use of the platform independent ***VTIEIO*** call allows each interpreter to implement different routing algorithms tuned for each APEP. For this simple APEP a simplified version of the standard maze-routing algorithm [8] was implemented.

Data Transfer: After the interpreter has built the accelerator, input data from DRAM is moved into the local input buffer BRAMS using ***VTIEIO*** (step 5, 6). The outputs of the accelerator are transferred from the output BRAM buffer back into DRAM using an additional call ***VAM_DMA*** (not shown in Figure 5.4) .

Control Operations: The ***VSTART*** (step 7, 8) directs the interpreter to initiate the execution of the array. ***VDONE*** (step 9, 10) returns status from the accelerator. ***VDEL*** (step 11, 12) directs the interpreter to release resources. Those steps are not presented on the left part of Figure 5.4.

5.5 Experimental Results and Analysis

All experiments reported in this section were performed on a reconfigurable cloud computer developed by Micron. Figure 5.5 shows a simplified block diagram of Micron's reconfigurable cloud computer. The host system is a dual Intel Xeon-2650 processor with 192GBs of DDR4

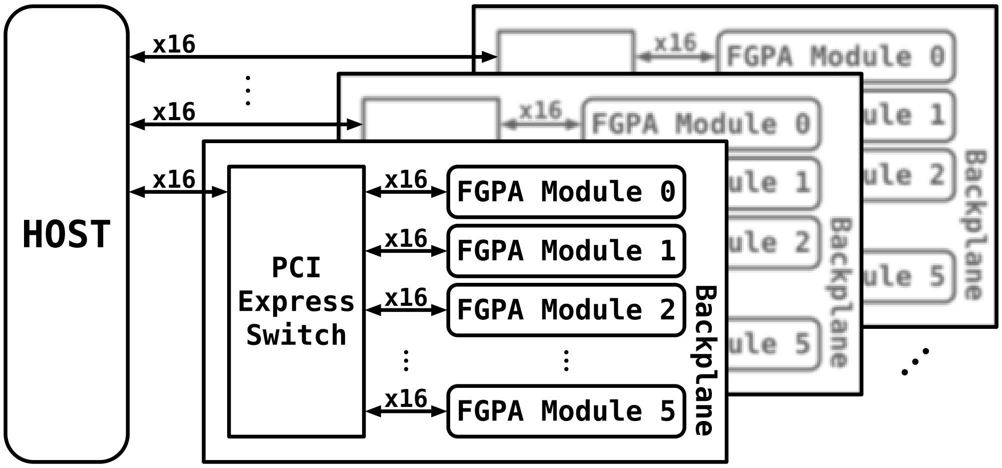


Figure 5.5: Block Diagram of Micron's Architecture [9].

memory. The system runs Ubuntu 14.04 with Linux kernel 3.12.0. On the backplane, a central PCI Express switch is used to connect the FPGA boards. Each backplane is an independent PCI Express device connected to the host. The system is contained in a 4U server with 6 backplanes. Each backplane connects six Xilinx Kintex 7 FPGAs.

All reported results are actual run time results. Low level execution times were gathered from a free running counter built into our overlay. End to end, or wall clock times that include overhead of protocol stack running on the Xeon were measured with the *gettimeofday()* Linux utility.

Software and hardware executables were generated using the design flow shown in Figure 6.6. Hardware versions of our selected programming patterns were generated using Vivado HLS. After generation, all executables were uploaded to the Micron reconfigurable cloud and executed. In addition to our interpreter, our software protocol stack running on the Xeon included pthreads, Linux and the Pico-Framework from Micron. The Pico-Framework provided PCIe protocols for data transfer between the Xeon, DRAM, and FPGAs.

Our base experimental system contained the APEP shown in Figure 5.3. This APEP was replicated within all 23 FPGAs. Each APEP was configured with a 1×4 array of PR tiles. Thus, our base experimental system contained a total of 92 tiles. Each tile contained 15,600 LUTS, 60

RAMB18, and 60 DSPs.

Our approach is based on the premise that the run time system can assemble presynthesized programming patterns within our PR overlay. To evaluate the feasibility of the approach, we identified the following three common types of programming patterns that would be useful within a shared library for a large cross section of data center programmers. These programming patterns do not represent any one specific application, but key computations that might need acceleration across many application domains. Each programming pattern was written in C and passed through Vivado HLS to generate bitstreams. Resource utilizations for the programming patterns are reported in Table 5.1.

- $\vec{V} = \sum[(\vec{A}+\vec{B}) \times \vec{C} - \vec{D}]$ The $\vec{V}$ was created using programming patterns for vector multiplication (*VMUL*), addition (*VADD*), subtraction (*VSUB*) and reduce (*REDU*). Figure 5.6(a) shows how the interpreter mapped these patterns into a four PR tiles. For our test cases, we used vectors of length 5.2 million (20MBytes).

- Correlation (**CORR**). This test shown in Figure 5.6(c) was created using the *AVG*, *SVSUB* and *DMUL* programming patterns. Figure 5.6(c) shows the interpreter mapped these patterns into 8 PR tiles. This tested the interpreters ability to map a single accelerator into tiles on different FPGAs. The **CORR** accelerator operated on vectors of single precision floating point values.

- DES Encryption and Decryption (**DES**). This test shown in Figure 5.6(b) represents an accelerator that fits into a single tile. The **DES** core encodes 24 Bytes of plaintext with a fixed encryption key. The 64 bit plaintext is sent to the accelerator through dual 32 bit input ports.

App.	PR Tile Utilizations (%)				LOC		Synthesis	Breeze
	LUTs	FFs	BRAMs	DSPs	HLS	IC	Time (*min*)	(*s*)
$\vec{V}$	0.22	0.22	0	5	35	13	20	$\leqslant 5$
CORR	3.80	2.03	53	8	90	11	30	$\leqslant 5$
DES	9.49	3.74	30	0	150	11	40	$\leqslant 5$

Table 5.1: Utilization of Resource for Applications

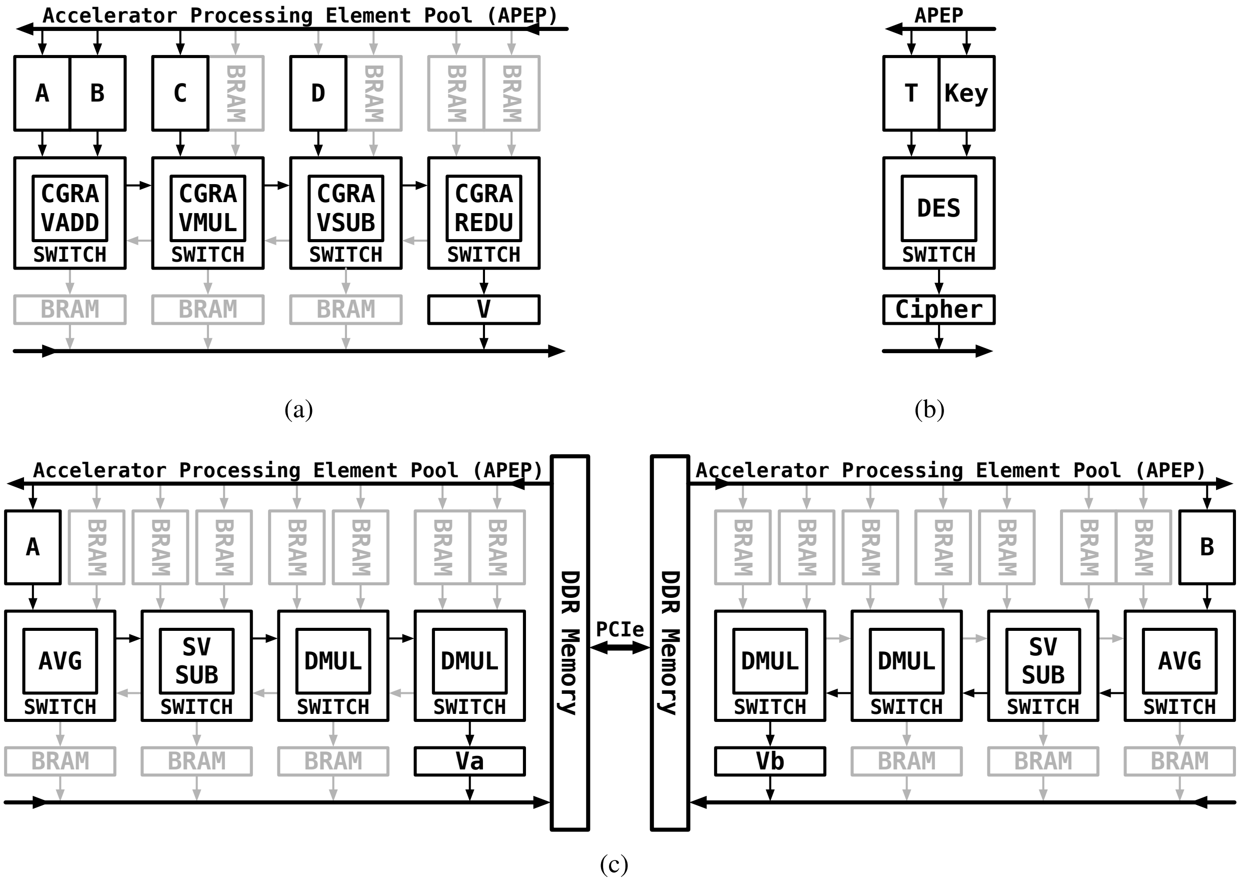
5.5.1 Discussion: Programmer Accessibility to FPGAs

First and foremost the experiments showed that our approach allowed accelerators to be compiled and assembled at run time by the interpreter. The three different tests showed how programming patterns could be written by a data center programmer without any knowledge of the underlining FPGA. The interpreter operated transparently to map the programming patterns into a single tile for **DES**, multiple tiles on the same FPGA for $\vec{V}$ and across multiple FPGAs for **CORR**. In effect, the use of our design flow, interpreter and overlay virtualized FPGA resources under a standard compilation flow.

Our flow also showed that synthesis can be removed from the data center programmers design flow. For illustrative purposes, we include the compile time versus the time that would be required to synthesize an equivalent accelerator on the right of the Table 5.1. The reported synthesis times for each benchmark is for the accelerator only. Although difficult to quantify, the difference in design times between composing programming patterns in software and specifying functionality using HLS would be significant.

5.5.2 Discussion: Scalability

One approach to increase performance within data centers is to parallelize software applications into concurrent threads. We wanted to evaluate the ability of the interpreter and PR overlay to support multiple threads, each of which may create an accelerator. Figure 5.7 shows the performance achieved by creating multiple threads that each contain accelerators.



(a)

(b)

(c)

Figure 5.6: Mapping three benchmarks on APEP.

The **CORR** required 8 PR tiles per threads. With 92 tiles available, up to 11 concurrent threads with accelerators could be run. The $\vec{V}$ and **DES** were tested with up to 16 concurrent threads. The resources did not limit the number of concurrent threads to 16. However, 16 threads were sufficient to evaluate the scalability of the interpreter and overlay.

The speedup of **DES** shows a nice linear relationship. Although the speedups for $\vec{V}$ and **CORR** are close to linear, they do start to degrade as the number of threads increases over 4. This degradation relates to the ability of the distributed PR tiles to saturate the bandwidth limitations of the PCIe. This effect has been previously reported in [9].

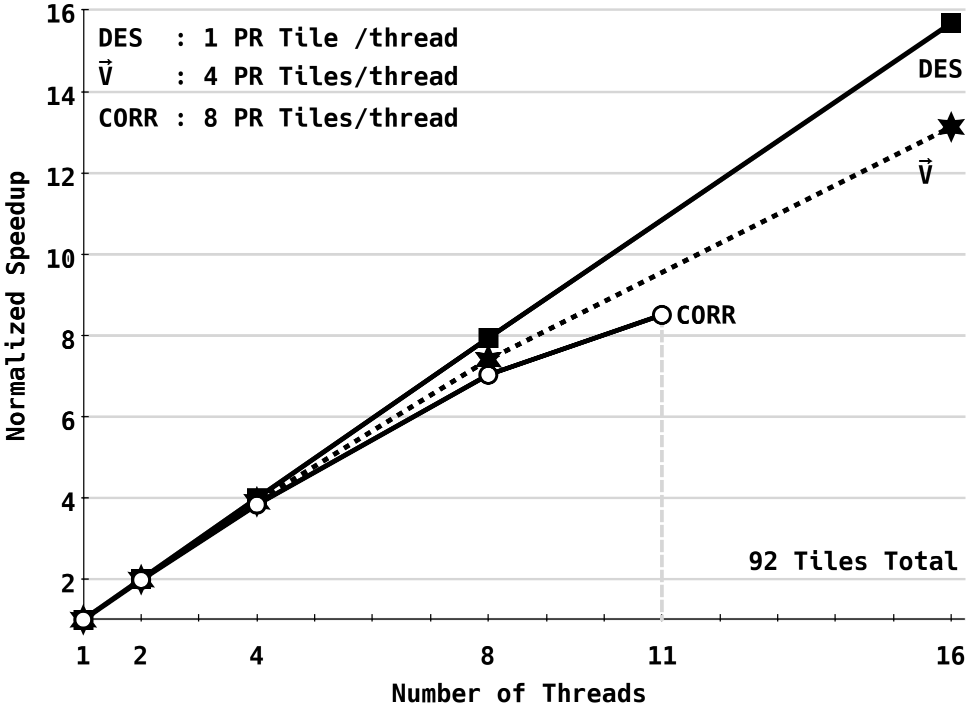


Figure 5.7: Scalability of the Overlay.

5.5.3 Discussion: Virtualization for Multiple Threads

Our final evaluation shows how the interpreter can manage all FPGAs as a virtual accelerator pool that can be shared by concurrent applications. Figure 5.8 shows a timeline of how the interpreter mapped and scheduled accelerators for our three applications. In this evaluation, each application was modeled as a number of threads. The $\vec{V}$ application was modeled as 8 threads which required 32 tiles. The **CORR** application was modeled as 4 threads requiring 32 tiles, and **DES** was modeled as 60 threads requiring 60 tiles. All three applications were started simultaneously. As shown in Figure 5.8, the interpreter assembled and ran the $\vec{V}$ and **CORR** accelerators but postponed running the **DES** accelerator. This delay resulted from the limitation of 92 tiles, as 124 tiles total would have been required. The interpreter manages its list of free tiles within a critical region protected by a mutex. The interpreter suspended when it could not allocate all the tiles required by the **DES** application. When the $\vec{V}$ finished, it signaled the interpreter, which then allowed the interpreter to reclaim and reassign the freed tiles to the **DES** application. This

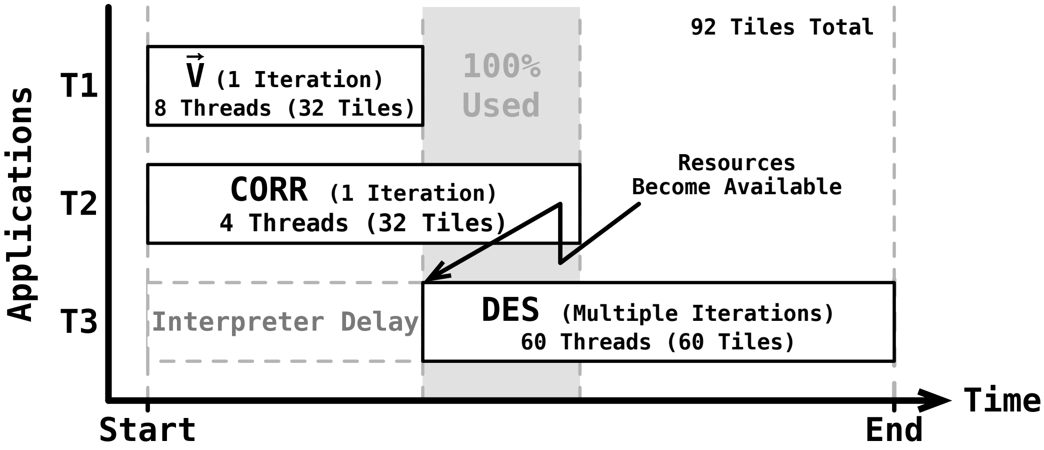


Figure 5.8: Simultaneously Sharing the Cloud Computing System with JIT approach among multiple applications.

experiment validated the interpreter to be reentrant and manage all resources as a virtual accelerator pool between concurrent applications. However, the scheduling algorithm was simplistic and is an area study for future research.

5.6 Conclusion

In this paper, we presented a new design flow, run time interpreter, and overlay that enables data center programmers to create accelerators using standard software development flows. Our approach eliminates the need for data center programmers to understand hardware design, use CAD tools, and pass designs through synthesis. Our experimental analysis confirms how the use of an interpreter and PR tile overlay can abstract multiple FPGAs into a virtual accelerator pool. This allows multiple threads to create accelerators that run concurrently within and across multiple physical FPGAs.

Acknowledgment

This work was in part sponsored by Micron Corporation. We thank Micron for access to their reconfigurable cluster.

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. *IEEE Micro*, 35(3):10–22, May 2015.

[2] Timothy Morgan. Intel Mates FPGA With Future Xeon Server.

[3] S. Byma, J.G. Steffan, H. Bannazadeh, A. Leon-Garcia, and P. Chow. Fpgas in the cloud: Booting virtualized hardware accelerators with openstack. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 109–116, May 2014.

[4] Sen Ma, Zeyad Aklah, and David Andrews. Just in time assembly of accelerators. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 21-23, 2016*, pages 173–178, 2016.

[5] N. George, Hyoukjoong Lee, D. Novo, T. Rompf, K.J. Brown, A.K. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne. Hardware system synthesis from domain-specific languages. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8, Sept 2014.

[6] Jason Yu, Guy Lemieux, and Christpher Eagleston. Vector Processing as a Soft-Core CPU Accelerator. In *FPGA '08: Proceedings of the 16th international ACM/SIGDA Symposium on Field Programmable Gate Arrays*, pages 222–232, New York, NY, USA, 2008. ACM.

[7] Hyouk Joong Lee, K.J. Brown, A.K. Sujeeth, H. Chafi, K. Olukotun, T. Rompf, and M. Odersky. Implementing domain-specific languages for heterogeneous parallel computing. *Micro, IEEE*, 31(5):42–53, Sept 2011.

[8] C.Y. Lee. An algorithm for path connections and its applications. *Electronic Computers, IRE Transactions on*, EC-10(3):346–365, Sept 1961.

[9] S. Gao and J. Chritz. Characterization of opencl on a scalable fpga architecture. In *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pages 1–6, Dec 2014.

Chapter 6

Archborn: A Custom Multiprocessor Architecture Generation Framework for Platform FPGAs

Sen Ma, Hongyuan Ding, Miaoqing Huang, and David Andrews

Abstract— Modern platform FPGAs are sufficiently dense to allow the assembly of a complete chip heterogeneous multiprocessor systems on chip (CHMPs) within a single die. Based on CHMP, every research group that sets out to explore how an application can be accelerated on an FPGA platform must firstly integrate processors, buses, memories, and IP components into a base architecture prior to beginning their application specific analysis. Lacking computer architecture background, many application developers will spend significant amounts of valuable research time to create and debug the base CHMP system. In this paper, we present Archborn, an open source tool that automates the generation of modifiable CHMP architectures. Archborn can be used by application developers to create a base CHMP system that can be synthesized in seconds. The systems created by Archborn can also be modified by those who wish to create fully customized CHMPs systems. We demonstrate the ease of use and the flexibility of Archborn by automatically generating two unique systems: a NUMA architecture that is modified with an Hthreads hw/sw co-designed microkernel for multithreading, and a NUMA architecture to support the OpenCL programming model. We present analysis on resource utilizations and scalability by creating systems with up to 64 processing elements (PEs).

6.1 Introduction

Field Programmable Gate Arrays (FPGAs) are continuing to grow in their use within academic laboratories. Application developers and computational science researchers routinely deploy FPGA to enable new applications in real time and distributed systems. While FPGAs provide significant advantages, individual research groups are spending unnecessary time and effort to build infrastructure in their local laboratories. It is not uncommon for students to spend six months to gain mastery of the tools before they even begin investigating their research questions. Additional months of effort can then be expended on learning advanced computer architecture topics, followed by hand building, integrating, and testing a base system. All of these efforts represent burdens of time and effort that could be eliminated through automation and sharing. In the absence of these benefits it is not uncommon for students to become frustrated with the breadth of knowledge they must learn before they can push their research. In some cases this has caused students to switch topics to more software based research to shorten their time to graduation. This situation unfortunately is not an isolated incident and is occurring across many research groups.

In the larger picture localized development does not support good science. Common sets of benchmarks, development tools and platform infrastructures are the basis for enabling fair and unbiased comparisons. Anecdotally, conference program committees are routinely being asked to evaluate the relative merits of a new FPGA based design or application without the benefit of good solid comparisons against prior art. Achieving such comparisons would require at least a doubling of engineering and development effort to rebuild the prior artifact.

It is our belief that creating base architectures, configuring multiple tool chains, and repetitive engineering design efforts can and should be automated. Automating this type of effort will enable students and researchers to address scientific questions quicker, and with an enhanced ability to increase experimentation across a broader range of configurations.

This paper presents Archborn, an open source tool for custom multiprocessor architecture generation based on Xilinx TCL routines in Vivado. Firstly, Archborn contains a library of hight-level APIs that can be used to create custom FPGA based Chip Heterogeneous Multiprocessor systems (CHMPs). Secondly, Archborn includes standard templates for automatically creating Symmetric Multiprocessor (SMP), Non-Uniform Memory Access (NUMA) and Partial Reconfigurable Accelerator (PRA) architectures. Users can then combine the use of the APIs and standard template interfaces, to automate the creation of new emerging CHMP architectures. To present the capabilities of Archborn, we show how the APIs and standard templates can be used to include additional support for the multithreading [1] and OpenCL [2] programming models. Specific contributions of this paper include:

- *Archborn APIs.* Sequences of low-level TCL commands with the same functionality are wrapped into a single routine to generate PEs, buses and memory components respectively. Archborn APIs are defined as open-source vendor neutral representations that can be implemented through vendor specific low-level TCL routines. Users can change the detailed definition of each API to fit in different FPGAs from various vendors through the TCL interface.

- *Archborn Templates.* To simplify the CHMP generation flows, Archborn bundles various APIs together to compose multiprocessor architecture templates such as SMP, NUMA, RING and PRA. A specific architecture is automatically built through a few basic configuration parameters, e.g., the type of Archborn template, amount of slave groups, the number of PEs on each groups and the size of shared memory. Users can modify the existing templates or compose new Archborn templates to meet their specific requirements.

- *Design Flow for Rapid Prototyping.* Based on the built in Archborn templates, a user can create and connect custom platform specific IP components to multi-level buses by using the returned bus list object. Therefore, designers are able to rapidly prototype custom hardware platforms with support for different programming models on FPGAs.

The remainder of this paper proceeds as follows. The next section provides an overview of background and related work. In Section 6.3, we first show how Vivado TCL primitive operations can be encapsulated into high-level Archborn APIs and Archborn template interfaces. Then, we introduce an example flow to generate a ring architecture. Section 6.4 presents how to use Archborn templates to rapidly prototype hardware platforms supporting different programming models. Evaluation results are then given on two specific systems: Hthreads and HOpenCL.

```
BEGIN microblaze
 PARAMETER INSTANCE = microblaze_1    # Processor Name
 PARAMETER HW_VER = 8.50.c            # IP Version
 PARAMETER C_DEBUG_ENABLED = 1        # Debug Enable
 PARAMETER C_USE_ICACHE = 1           # Enable I$
END
```

(a) MHS Descriptions for a Processor Instance.

```
create_bd_cell \                      # TCL Command
 -type ip \
 -vlnv xilinx.com:ip:microblaze:9.5 \ # IP Version
  microblaze_1                        # Processor Name

set_property \                        # TCL Command
 -dict [list CONFIG.C_USE_ICACHE {1}]\# Enable I$
  [get_bd_cells microblaze_1]
```

(b) TCL Commands to Create and Configure a Processor Instance.

Figure 6.1: MHS Descriptions and TCL Commands.

Finally, conclusions with a discussion on the additional research are addressed.

6.2 Background and Related Work

Automating the creation of a CHMPs system makes it convenient for researchers to explore various problems related to high-performance computing (HPC) [3], fault tolerant and self-healing systems [4], exploring the performance versus cost tradeoffs for custom accelerators within embedded systems [5] [6], and engaging in prototyping and design space exploration [7, 8].

Xilinx Platform Studio (XPS) helps hardware designers build, connect and configure systems with embedded processors. The MHS file serves as an input to the Platform Generator tool. Later, according to MHS file synthesis tools can generate the hardware description language (HDL) definition of the system. In an MHS file, IP definitions start with `BEGIN` and terminate with `END`. The key word `PARAMETER` is used to configure parameters of the processor. Designers can change the configurations of the processor by modifying the values of parameters. The GUI does not support the generation of CHMP with more than two processors. To create a system with more than two processors, designers have to manually duplicate the configuration descriptions in MHS files.

MHS files are utilized by several research projects to create their automated system assembly methods targeting Xilinx devices. Hthreads-Cloud [1] outlined a cloud-based tool flow to automatically create complete CHMPs systems for platform FPGAs. In [9], the authors designed a flow for developing application-specific platforms by automatically generating CHMPs systems for different applications.

Vivado, the latest Xilinx tool suite, provides extensive functionalities for all programmable platform FPGAs. Designing a CHMPs system in Vivado differs from that in XPS in at least three ways. Firstly, it discontinues the support of MHS files for configurations. Secondly, TCL commands are used to create and configure all components in a CHMP design. Last but not the least, it provides the interface to import a set of TCL commands as a script file to automate the design. Compared with MHS files, Figure 6.1(b) presents the TCL commands to create and configure a processor instance. The command `create_bd_cell` is used to create the processor and `set_property` command is used to configure the component.

Although using the TCL shell interface in Vivado is convenient, using low-level TCL routines to create a component and configure its properties into a CHMP system can be a tedious and error-prone task. Besides, the whole design flow requires a detailed low-level knowledge of architectural configurations and interconnects. To offload this burden from designers, a common TCL extension library for Vivado is needed. TincrCAD [10] provided a TCL extension library to assist users in the creation of custom CAD tools.

Categories	API Names	Semantics & Descriptions
PEs	pe_gen	***pe_gen*** {*key pe_type pm_size ic_size dc_size stream*} *Creating objects of processing elements.*
AXI Buses	bus_gen	***bus_gen*** {*key n_master_port n_slave_port*} *Generating AXI bus interconnections.*
	bus_connect	***bus_connect*** {*master m_port slave s_port*} *Connecting two AXI ports from master to slave.*
	bus_module_gen	***bus_module_gen*** {*key ip_type_type addr_range*} *Creating AXI compatible IP_type modules.*
Stream Interface	stream_gen	***stream_gen*** {*key n_master_port n_slave_port*} *Generating stream bus interconnections.*
	stream_module_gen	***stream_module_gen*** {*key ip_type*} *Creating stream compatible IP modules.*
	stream_connect	***stream_connect*** {*master m_port slave s_port*} *Connecting two stream ports from master to slave.*
PR	pr_module_gen	***pr_module_gen*** {*key ip_type f_dcp addr_range*} *Creating IP modules for partial reconfiguration.*
Memories	mem_gen	***mem_gen*** {*key mem_type addr_range*} *Creating memory module based on mem_type.*
Templates	template_gen	***template_gen*** {*Arch_type nGroup nPE mem_size*} *Creating template, e.g. SMP, NUMA, RING, PRA.*

Table 6.1: Archborn APIs and Templates.

6.3 Archborn Framework

The Vivado TCL shell provides a basic set of routines to create and configure IP components. TCL scripts shown in Figure 6.1(b) provide an alternative way to release the constrain of MHS formats. However, using low-level TCL routines will involve many details for system designers and may cause distraction. Designers may shift the attention from the pure architecture design to tedious details such as bus configurations, interconnections and even memory space allocation. Designers will benefit from bundling TCL routines into a script file for redundant design in the future. However, such TCL scripts lack the flexibility and portability. Further, due to the limitation of Vivado TCL shell, it cannot create a custom architecture using different

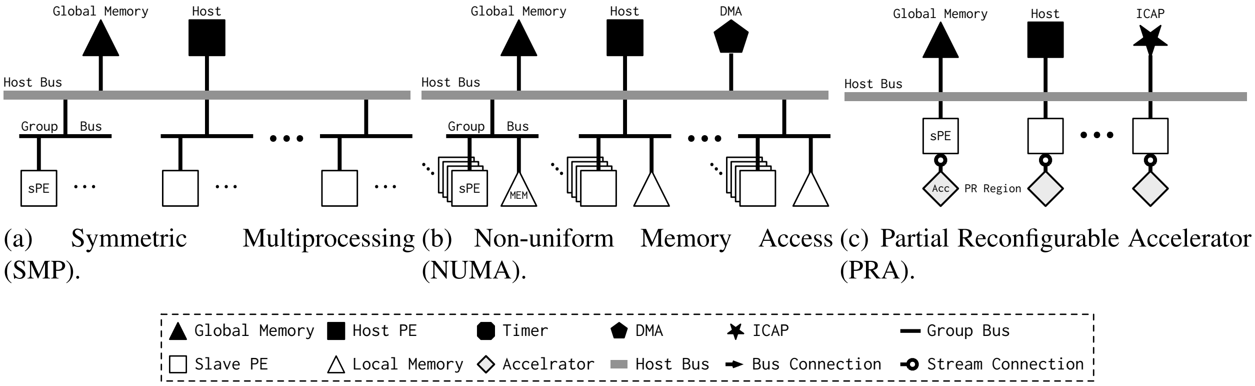
```
proc pe_gen {key pe_type pm_size ic_size dc_size stream}\
{
  if {$pe_type == mb} {
    set pe_name mb_$key
    create_bd_cell\
      -type ip -vlnv xilinx.com:ip:microblaze:9.5\
        $pe_name
    set_property -dict [list CONFIG.C_FSL_LINKS $stream]\
      [get_bd_cells $pe_name]
    if {$ic_size == 0} {
      set_property -dict [list CONFIG.C_USE_ICACHE {0}]\
        [get_bd_cells $pe_name]
    } else {
      set_property -dict [list CONFIG.C_USE_ICACHE {1}]\
        [get_bd_cells $pe_name]
      set_property\
         -dict [list CONFIG.C_CACHE_BYTE_SIZE $ic_size]\
          [get_bd_cells $pe_name]
    }
    if {$dc_size == 0} {
      set_property -dict [list CONFIG.C_USE_DCACHE {0} ]\
        [get_bd_cells $pe_name]
    } else {
      set_property -dict [list CONFIG.C_USE_DCACHE {1}]\
        [get_bd_cells $pe_name]
      set_property\
        -dict [list CONFIG.C_DCACHE_BYTE_SIZE $dc_size]\
          [get_bd_cells $pe_name]
    }
  } else { ... } # Code trimmed due to limited space
  return $pe_name
}
```

Figure 6.2: Implementation of a Representative Archborn API.

programming models, such as Hthreads and HOpenCL. To assist system designers in creating CHMP, Archborn extracts an abstraction layer above the Vivado TCL interfaces. By targeting various programming models, it helps system designers improve design productivity and reduce complexities to generate CHMP on modern platform FPGAs.

6.3.1 Archborn APIs

Each Archborn API is a group of TCL commands that are categorized into the same functionality. As an example shown in Figure 6.1(b), TCL commands `create_bd_cell` and `set_property` are paired to instantiate one processor and configure it with custom parameters. By using Archborn APIs, designers will be released from setting up component configurations through many mouse clicks and from detailed TCL commands with trivial primitive parameters.



(a) Symmetric Multiprocessing (SMP).

(b) Non-uniform Memory Access (NUMA).

(c) Partial Reconfigurable Accelerator (PRA).

Figure 6.3: Architecture Demonstration of Different Archborn Templates.

Table 6.1 lists essential Archborn APIs categorized by functionalities and wrapped into single cohesive routines. As shown in Figure 6.2, take **`pe_gen`** as an example, it is defined as a TCL procedure by using TCL keyword `proc`. Six parameters are required for this API. The API routine will first create an object name concatenating the type of object and the value of parameter *key* as a suffix. *type* stands for the type of the processor (e.g., MicroBlaze or ARM). *pm_size*, *ic_size* and *dc_size* indicate the sizes of private memory, instruction cache and data cache, respectively. The last parameter, *stream*, is the number of streaming interfaces when the processor is configured as a MicroBlaze.

Archborn APIs also support the generation of partial reconfiguration (PR) modules. Previously, in order to generate a PR module in a CHMP, designers have to manually define PR regions and associate its region to a specific component, which is called a PR constrain. By using Archborn PR APIs that are shown in Table 6.1, designers can call **`pr_module_gen`** to generate a PR module whose PR constrains will be automatically appended to the Xilinx Design Constraint (XDC) file.

6.3.2 Archborn Templates

After defining Archborn APIs, we further design several Archborn templates to build common hardware architectures. Figure 6.3 shows three of them. Within a symmetric multiprocessing

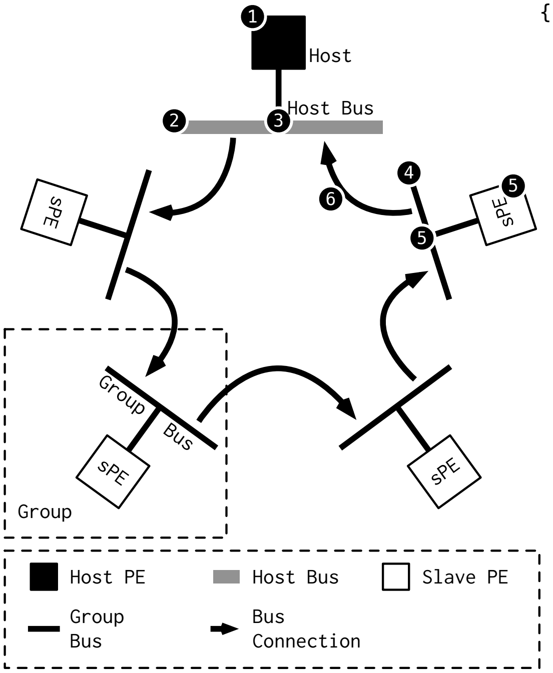
(SMP) architecture shown in Figure 6.3(a), multiple `Group Bus` can be connected to `Host Bus` and several `Slave PE` are attached to each `Group Bus`. A non-uniform memory access (NUMA) architecture is shown in Figure 6.3(b). Different from SMP architecture where groups contains only PEs, in NUMA architecture a group contains an extra group memory. The last architecture demonstrated in Figure 6.3(c) is a partial reconfigurable accelerator architecture (PRA). Each hardware accelerator is defined as a partial reconfigurable region and connected to its corresponding PE through streaming interfaces. Next we use the Ring architecture shown in Figure 6.4 to demonstrate how to construct an Archborn template.

Creation of Host Processor and Bus

Although the `HOST` PE is not necessary for every CHMP design, all architecture templates in this work have a `Host` PE for a better understanding. For each **`pe_gen`** (steps 1, 5), Archborn API will create a PE object following parameters indicated in Table 6.1. The name of the PE object will be returned. For instance, in the step 1, a `Host` PE will be created. It is configured as a MicroBlaze with 4KB private memory and without caches (data and instruction ones) or streaming ports. The object name of `Host` PE will store in the variable *Host*. By using *$host*, the name of `Host` PE can be accessed, which is presented in step 3. An interconnection bus for `Host` PE is created by using **`bus_gen`** API (step 2) and the name is saved in the variable *host_bus*. In step 3, the **`bus_connect`** will connect `Host` PE with `Host Bus`.

Creation of Slave Processors and Buses

After the `Host` PE and bus are created, `Slave` buses and PEs can be created in steps 4 and 5, respectively. Since in this example, there are four groups, each containing a `Slave` PE and a bus, a TCL loop is used to generate all groups. Within each loop, a list of PE objects and a list of bus objects will be returned. The variable *bus_list* contains all names of bus objects and the variable *pe_list* includes all `Slave` PE objects. Since `Host` PE has already been connected to the `Host`



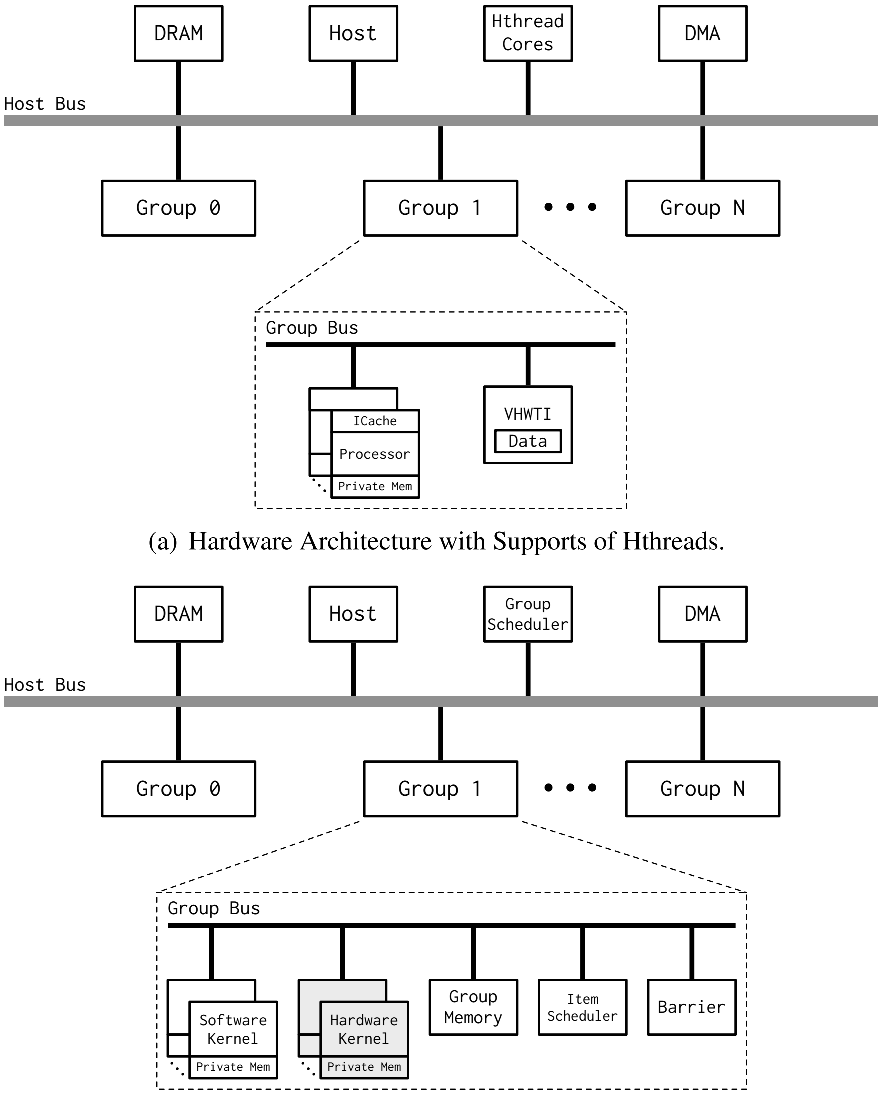
```
proc interface_gen {type nGroup nPE mem_size} \
{
   set nTotal [expr $nGroup+1]
   if {$type == RING} {
    ❶ set host [pe_gen host mb 4 0 0 0]
    ❷ set host_bus [bus_gen host_bus 1 2]
    ❸ bus_connect $host 0 $host_bus 0
    ❹ set bus_list {}
      lappend bus_list $host_bus
      for {set i 0} {$i<$nGroup} {incr i} {
       lappend bus_list [bus_gen g_$i 2 2]
      }
    ❺ set pe_list {}
      for {set i 0} {$i<$nGroup} {incr i} {
        for {set j 0} {$j<$nPE} {incr j} {
          lappend pe_list \
            [pe_gen $j mb 4 0 0 0]
          bus_connect [lindex $pe_list $j] 0
            [lindex $bus_list $i] [expr $j]
        }
      }
    ❻ for {set i 0} {$i<$nTotal} {incr i} {
        bus_connect [lindex $bus_list $i] 0 \
          [lindex $bus_list [expr $i+1]] 1
      }
   } else { ... } # Code trimmed
   return $bus_list
}
```

Figure 6.4: A Template of the Ring Architecture Generation Flow.

Bus, it is not included in the list of PEs. Within the nested loop in step 5, the inner loop produces PEs in each group and the outer loop connects each PE with its corresponding Group Bus.

Architecture Connection

All group buses and host bus will be connected appropriately in step 6 to create a ring structure. In this example, the ring structure is connected following the anticlockwise direction. In order to implement a bi-direction topology, another TCL loop can be used by swapping parameters of **bus_connect**.



(a) Hardware Architecture with Supports of Hthreads.

(b) Hardware Architecture with Supports of HOpenCL.

Figure 6.5: Hardware Platforms of Two Programming Models.

6.4 Hardware Platforms with Programming Models

6.4.1 Targeted Programming Models

On platform FPGAs with a CHMP, various parallel programming models are presented to manage hardware resources. On the one hand, thread level parallelism (TLP) and data level parallelism (DLP) can be extracted; on the other hand, programming models unify the gap between hardware and software design. Parallel programming models provide interfaces for both software programs and hardware accelerators. In this section, Hthreads [1] and HOpenCL [2] are used as examples to demonstrate the abilities of Archborn to generate specific hardware platforms with different programming models.

Hthreads

Hthreads is a pthreads compliant hardware microkernel to provide a unified set of operating system services for application programs running on scalable numbers of parallel heterogeneous processor resources. Figure 6.5(a) shows the overview of the hardware platform for an Hthreads system. On host bus, Hthreads cores consist of supporting modules of thread manager, thread scheduler, and synchronization manager, and conditional variables.

The Virtual Hardware Thread Interface (VHWTI) is an abstraction layer for threads residing in custom hardware accelerators and slave processors respectively, to seamlessly interface into the Hthreads system. This allows portable HDL or general purpose processors to interface simplistically with Hthreads via several read/write registers. The VHWTI operates in coordination with a small kernel (4KB) called the Hardware Abstraction Layer (HAL). The HAL transforms Hthreads system calls into simple load/store operations that access the Hthreads cores. Using general purpose processors as replacement components for custom hardware circuits offers the flexibility and the increased design productivity but at the expense of performance.

HOpenCL

OpenCL is a framework to design parallel applications on various computation resources (e.g., CPUs and GPUs). The current OpenCL specification is heavily influenced by GPU programming. In [2], the author presented a hybrid hardware platform combining with OpenCL-flavor programming model called HOpenCL. In this work we further simplify the design flow by using the Archborn generation framework. Figure 6.5(b) demonstrates the hardware platform generated by Archborn for the OpenCL programming model. Similar to the Hthreads hardware platform,

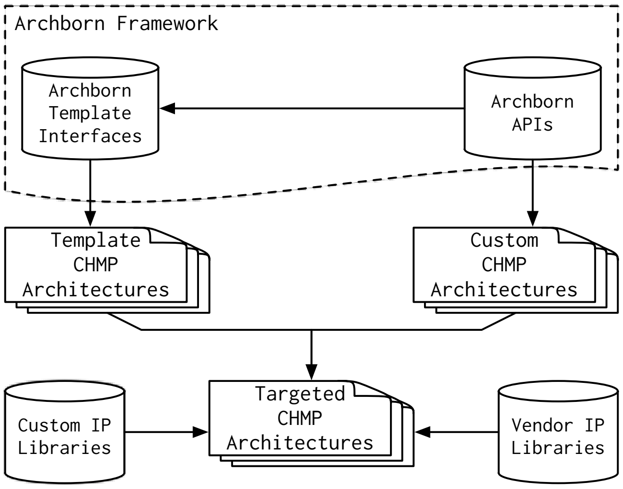

Archborn Framework
Archborn Template Interfaces
Archborn APIs
Template CHMP Architectures
Custom CHMP Architectures
Custom IP Libraries
Targeted CHMP Architectures
Vendor IP Libraries

Figure 6.6: Design Flow of Archborn Framework.

computation resources are divided into multiple groups. Inside each group, multiple software OpenCL kernels run on PEs, and hardware ones execute as dedicated hardware accelerators. Private, group and global memories form the three-level memory hierarchy in order to meet the OpenCL specification. Group scheduler dispatches group partitions of one OpenCL kernel to each hardware group of computation resource. Within each hardware computation resource, item scheduler decomposes the corresponding task group into items and arranges them to software or hardware kernels for execution. Barriers provide the synchronization insides each group.

6.4.2 Design Flow with Archborn

As shown in Figure 6.6, design flows to generate multiprocessor architectures with Archborn are demonstrated in details. A custom multiprocessor architecture can be generated by using Archborn APIs. Archborn templates provide a more convenient method and interface to build template architectures (e.g., RING, NUMA, and SMP). A targeted multiprocessor architecture can be built with either Archborn APIs or Archborn templates. The Archborn generation flow is based on bus hierarchies to expand the system. TCL codes shown in Figure 6.7 demonstrate the generation TCL scripts of Hthreads and HOpenCL platforms by using Archborn templates as the interface. `template_gen` will return a bus list object where the host bus and

```
set bus_list [template_gen NUMA 6 4 4KB]
bus_connect [lindex $bus_list 0] 3 \
    [bus_module_gen t_manager thread_manager 4KB] 0
bus_connect [lindex $bus_list 0] 4 \
    [bus_module_gen t_sche thread_scheduler 4KB] 0
bus_connect [lindex $bus_list 0] 5 \
    [bus_module_gen sync_manager sync_manager 4KB] 0
bus_connect [lindex $bus_list 0] 6 \
    [bus_module_gen con_var con_variable 4KB] 0
for {set i 0} {$i < 6} {incr i} {
    bus_connect [lindex $bus_list [expr $i + 1]] 4 \
        [bus_module_gen vhwti vhwti 4KB] 0
}
```

(a) TCL Scripts to Generate Hthreads Platform.

```
set bus_list [template_gen NUMA 6 4 64KB]
bus_connect [lindex $bus_list 0] 3 \
    [bus_module_gen g_sche group_scheduler 4KB] 0
for {set i 0} {$i < 6} {incr i} {
    bus_connect [lindex $bus_list [expr $i + 1]] 5 \
        [bus_module_gen i_sche item_scheduler 4KB] 0
    bus_connect [lindex $bus_list [expr $i + 1]] 6 \
        [bus_module_gen barrier barrier 4KB] 0
    for {set j 0} {$j < 4} {incr j} {
        bus_connect [lindex $bus_list \
            [expr $i + 1]] [expr $j + 7] \
            [pr_module_gen acc$i$j mm mm$i$j.dcp 4KB]
    }
}
```

(b) TCL Scripts to Generate HOpenCL Platform.

Figure 6.7: Hardware Platforms with Programming Models Generated by Archborn Templates.

# of PEs	Hthreads		HOpenCL	
	LUTs	BRAM	LUTs	BRAM
1	25,009 (8.0%)	36.0 (3.5%)	-	-
2	28,919 (9.5%)	45.5 (4.4%)	-	-
4	35,449 (11.7%)	65.5 (6.4%)	24,045 (7.92%)	71 (6.89%)
8	49,328 (16.3%)	105.5 (10.2%)	43,718 (14.4%)	136 (13.2%)
16	76,926 (25.3%)	185.5 (18.0%)	70,435 (23.2%)	230 (22.33%)
32	132,274 (43.6%)	345.5 (33.54%)	129,819 (42.76%)	374 (36.31%)
64	240,778 (79.3%)	665.5 (64.61%)	238,052 (78.41%)	723 (70.19%)

Table 6.2: Resource Utilization of Hthreads and HOpenCL Platforms With Various PEs

multi-level group buses are recoded. Custom IPs and other vendor-provided IPs are added to corresponding buses by iterating the bus list.

Both Hthreads and HOpenCL hardware platforms are inherited from the NUMA architecture. TCL scripts shown as examples in Figure 6.7(a) and Figure 6.7(b) generate six groups in each of which both Hthreads and HOpenCL platform contain four processing elements. The first element of the returned bus list refers to the host bus. The remaining ones are group buses. IP modules can be connected to host and group buses respectively. For HOpenCL platform shown in Figure 6.7(b), four hardware kernels as PR modules are added to each group bus. In this example, we use the matrix multiplication modules as hardware accelerators.

6.4.3 Experimental Results

Our intent for Archborn is not to reveal the best architecture for any application but to provide a rapid automated generation tool. In this section, we verified the correctness of Archborn by using Hthreads and HOpenCL frameworks. We also presented the creation of CHMP systems with different programming models. Based on Archborn template, a large base CHMP system is built rapidly. Researcher can extend or modify the base CHMP system for specific needs.

Table 6.2 shows the resource utilization of both Hthreads and HOpenCL frameworks built with Archborn templates. In this experiment, we use Xilinx tool chain Vivado with VC707 board. We evaluate various numbers of PEs ranging from 1 to 64. The AXI interconnections can accommodate up to 16 slaves or 16 masters modules. In order to hold as many as PEs, we can either add multi-level group buses or connect more PEs to each group bus. Since HOpenCL will utilize group memories for fast data access within one group, there is no need to have only one or two PEs within one group.

We use the 2-dimensional integer matrix multiplication as the benchmark to evaluate the scalability of the multiprocessor architectures with supports of Hthreads and HOpenCL, respectively. The inputs of our benchmark are two 512×512 matrices. The output result is also a 512×512 matrix. We divide the output matrix into multiple groups that can be arranged to every processing element for computation. Depending on whether the actual hardware architecture has group memories (e.g., if one group contains multiple PEs, these PEs can share one group memory), partial input data can be fetched into group memories by DMAs for fast access. Figure 6.8 demonstrates the scalability of generated Hthreads and HOpenCL platforms. The total number of PEs for each platform is from 4 to 24 with 4-stride. Both Hthreads and HOpenCL platforms demonstrate the steady and similar scalability as the number of PEs increases.

6.5 Conclusion

Modern platform FPGAs are capable of accommodating a complete CHMP system including general-purpose processors, vendor-provided IPs and custom hardware accelerators. For system architecture researchers, it is necessary to provide a multiprocessor generation tool for rapid validation and system generation. Besides, a unified interface to embed modern programming models is essential to explore both DLP and TLP in CHMP.

In this work, we present Archborn, an open source multiprocessor architecture generation tool based on TCL scripts on platform FPGAs, to unify different hardware multiprocessor architectures with various programming models for rapid system generation. According to bus

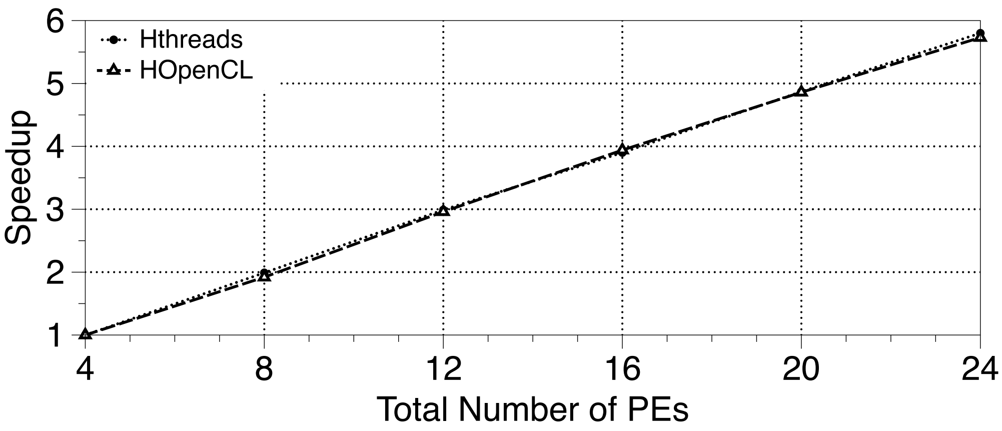


Figure 6.8: Evaluation of Scalability With Various PEs.

hierarchies within CHMP, we extract and wrap common operations for multiprocessor system generations as Archborn APIs. The Archborn templates simplify the generation flow to build common CHMP architectures. We take Hthreads and HOpenCL as targeted programming models to present and verify the correctness of Archborn. The resource utilization and scalability of CHMP with up to 64 PEs are evaluated. A matrix multiplication benchmark is parallelized and executed on platform extended from the built CHMP by using two programming models. As an open source tool, Archborn will be released on public repository.

Although Archborn can significantly improve the productivity to generate a CHMP, there are two limitations. Firstly, users need to be familiar with vendor-specific TCL libraries. Secondly, current Archborn APIs follow function programming manners to manage IP resources. In the future work, Archborn will introduce the object-oriented programming (OOP) method to allow users to build specific architecture and write application source code using the same OOP language, such as C++ or Java.

Acknowledgment

This work is supported in part by NSF under grant CNS-1219062.

References

[1] E. Cartwright, A. Fahkari, S. Ma, C. Smith, M. Huang, D. Andrews, and J. Agron. Automating the design of mlut mpsopc fpgas in the cloud. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 231–236, Aug 2012.

[2] Hongyuan Ding and Miaoqing Huang. A unified opencl-flavor programming model with scalable hybrid hardware platform on fpgas. In *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, pages 1–7, Dec 2014.

[3] D. Gohringer and J. Becker. Fpga-based runtime adaptive multiprocessor approach for embedded high performance computing applications. In *VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on*, pages 477–478, July 2010.

[4] M. Bucciero, J.P. Walters, R. Moussalli, Shanyuan Gao, and M. French. The powerpc 405 memory sentinel and injection system. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 154–161, May 2011.

[5] S. Ma, Z. Aklah, and D. Andrews. A run time interpretation approach for creating custom accelerators. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sept 2015.

[6] T.D.A. Nguyen and A. Kumar. Pr-hmpsoc: A versatile partially reconfigurable heterogeneous multiprocessor system-on-chip for dynamic fpga-based embedded systems. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–6, Sept 2014.

[7] S. Fernando, F. Siyoum, Yifan He, A. Kumar, and H. Corporaal. Mampsx: A design framework for rapid synthesis of predictable heterogeneous mpsocs. In *Rapid System Prototyping (RSP), 2013 International Symposium on*, pages 136–142, Oct 2013.

[8] R. Van Langendonck, A.K. Lusala, and J. Legat. Mpsocdk: A framework for prototyping and validating mpsoc projects on fpgas. In *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*, pages 1–8, July 2012.

[9] Sen Ma, Miaoqing Huang, and D. Andrews. Developing application-specific multiprocessor platforms on fpgas. In *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*, pages 1–6, Dec 2012.

[10] B. White and B. Nelson. Tincr - a custom cad tool framework for vivado. In *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, pages 1–6, Dec 2014.

Chapter 7

Conclusion

7.0.1 Summary

This thesis presented a new approach to enable programmers to use standard software development flows to create hardware accelerators. An important contribution of this work is the elimination of synthesis from the path of a programmers design flow. The approach introduced a new PR tile overlay and a set of interpreter calls that brings portability into the process. This will greatly facilitate the use of FPGAs within our software dominated information technology sector. We showed how equivalent accelerator functionality could be represented by composition, compilation and run time interpretation of functors. Results were presented showing a complete end to end capability and also show the costs in terms of additional resource overhead for the accelerator functionality as well as the overlay.

7.1 Future Work

Building the prototype raised just as many questions as it answered. First, the number and size of tiles used in the prototype were chosen somewhat arbitrarily, based on a small sample of functors running on a small tile array. The tile size was set to be sufficiently large enough to hold any of our prototype functors. The numbers of tiles were set based on estimating the number of functors needed to be composed our benchmark suite. Setting the geometry and interconnect of tiles needs a more quantitative treatment and should be automated. The Xilinx Vivado 15.2 PR flow was used to create the bitstreams for each primitive. This also needs to be automated within the front end DSL environment. Further, the bitstreams created for the prototype in Vivado are not relocatable. This required creating bitstreams for every primitive for each PR regions. New flows that enable relocatable bitstreams need to be included within the front end DSL environment. This will significantly reduce the hardware designers job of creating the primitives as well as provide a more portable solution. Our analysis does not include discussion of the different size of PR region. Additional investigations are needed to determine an efficient solution to turn a fully custom accelerator into a “macro” primitive and can be mapped across multiple tiles as if it were a unified custom component.

We anticipated that eliminating synthesis would come with some reduction in peak performance compared to a fully customized and synthesized accelerator. While we anticipated this reduction in performance, our small suite of benchmarks did not validate this cost. We conservatively interpret our results as simply not yet negating the feasibility of the approach. The approaches effect of performance needs significantly more study using multiple DSLs on different FPGA chips. Clearly, speedups and achievable performance will vary on an application to application basis. Finally, we performed a rapid prototype of the run time interpreter. This initial interpreter needs to be optimized to understand how the overhead of interpreting hardware accelerators compares with interpreting software.