Run Time Interpretation for Creating Custom Accelerators

Sen Ma, Zeyad Aklah, David Andrews
Department of Computer Science and Computer Engineering
University of Arkansas
Fayetteville, AR 72701, USA
Email: {senma, zaklah, dandrews}@uark.edu

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.

I. 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 programers 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.

A. Overview of Approach

Figure 1 motivates our approach through a simple example. At the top left of Figure 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 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 1. The functions contained within the programming patterns such as e$\times$e in *Map* and a + b in *Reduce* are not the actual code, but symbolic

978-3-9815370-6-2/DATE16/

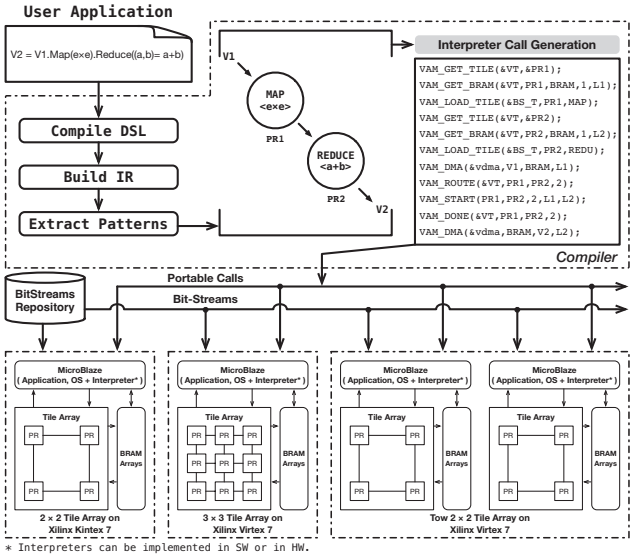


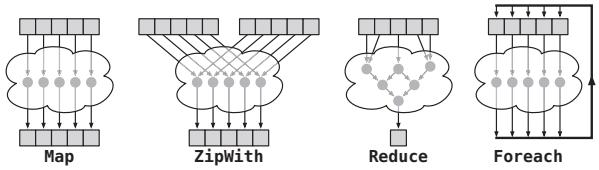
Fig. 1. Design Portability with JIT.

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 retarget the compiler to output a series of platform independent interpreter instructions as shown on the right in Figure 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 heterogenous systems by separating policy from mechanism. The interpreter commands are completely platform independent.

Figure 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 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



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}

Fig. 2. DSL Programming Patterns [7].

TABLE I. VAM CALLS

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.

partial reconfiguration. Figure 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 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.

II. 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]–[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

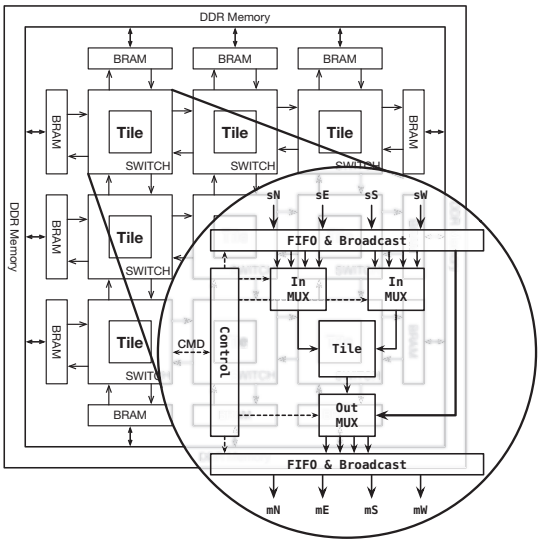


Fig. 3. 3×3 Tile Array and Interconnect Network.

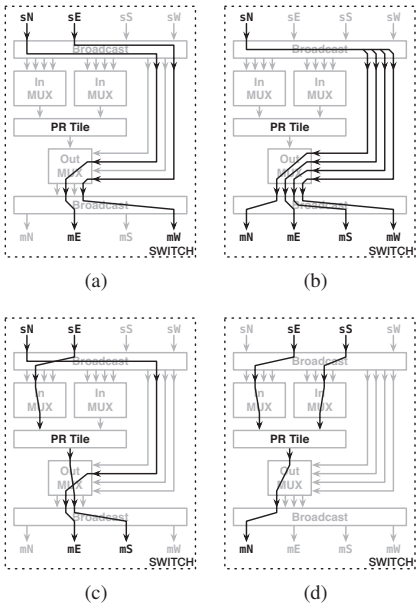


Fig. 4. Switch Routing.

achieved through compilation instead of synthesis on existing FPGAs [8]–[10].

We created the new type of overlay shown in Figure 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.

A. PR Tiles

The 2-D array shown in Figure 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 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.

B. Programmable Switch

Figure 4 provides an exploded view of a switch. Figure 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(a) shows how two inputs can be configured to pass data to two different outputs. Figure 4(b) shows the switches broadcast capability, which allows a single input to fan out data on multiple output routes. Figure 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(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.

C. 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.

III. Interpreter

Table I 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 1 where the same interpreter calls are executed on three different configurations of our overlay embedded within different FPGAs. The simple example in Figure 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 overaly. 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 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 route the accelerator at run time. The same separation of policy and mechanism bring portability over different intermediate fabrics.

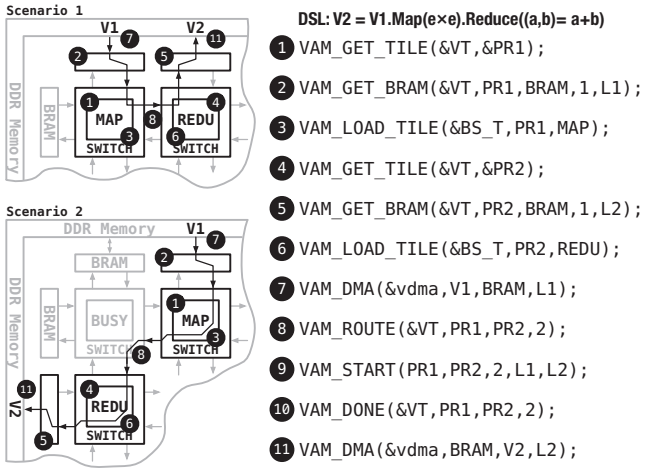


Fig. 5. VAM Call Flow With Different Cases.

IV. Interpreter Functionality

In this section we continue with our running example shown in Figure 1 to illustrate how the interpreter executes the calls on the two overlay configurations shown in Figure 5.

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. 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 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 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 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

TABLE II. PRE-SYNTHESIZED PARALLEL PATTERN SAMPLES

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

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.

V. EXPERIMENTAL RESULTS AND ANALYSIS

Figure 1 shows three unique systems created to evaluate our approach. The first system contained 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 Virtex 7 and contained two 2×2 overlays. In all cases the overlays were interfaced to a MicroBlaze processor as tightly coupled accelerators. Figure 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.

A. Creating the Accelerators

The first column in Table II 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.

TABLE III. PERFORMANCE

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	1.4s	-	-	-	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	1.4s	-	-	-	1×
	HW (Full Module)	1,046	-	63	983	1,331×
	HW(JIT)	840	39	63	738	1,659×

*Software running on MicroBlaze configured with data & instruction cache and floating point unit.
†Both software and hardware are running under 100MHz.
‡The overhead of the interpreter including VAM_GET_TILE(4μs), VAM_GET_BRAM(4μs), VAM ROUTE(31μs).

We used the patterns and expressions listed in Table II to create the four benchmark accelerators listed in Table III. 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 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 to be removed from their development paths.

B. Discussion: Performance Analysis

It was anticipated that run time assembling accelerators would suffer some measure of degraded performance com-

pared 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 III were generated on a Virtex7 on the 3×3 overlay. Thus we focus our discussion on the execution times reported in Table III.

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 III.

VI. 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.

1) 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. [Online]. Available: http://archive2.cra.org/uploads/documents/resources/rissues/computer.architecture_.pdf

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

[3] T. Morgan. Intel Mates FPGA With Future Xeon Server. [Online]. Available: http://www.enterprisetech.com/2014/06/18/intel-mates-fpga-future-xeon-server-chip/

[4] H. J. Lee, K. Brown, A. Sujeeth, H. Chafi, K. Olukotun, T. Rompf, and M. Odersky, "Implementing domain-specific languages for heterogeneous parallel computing," *Micro, IEEE*, vol. 31, no. 5, pp. 42–53, Sept 2011.

[5] N. George, H. Lee, D. Novo, T. Rompf, K. Brown, A. 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*, Sept 2014, pp. 1–8.

[6] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp, "Achieving Programming Model Abstractions For Reconfigurable Computing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 34–44, January 2008.

[7] A. Sujeeth, H. Lee, K. Brown, T. Rompf, H. Chafi, M. Wu, A. 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*, Oct 2010, pp. 13–22.

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

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

[11] J. Yu, G. Lemieux, and C. 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*. New York, NY, USA: ACM, 2008, pp. 222–232.

[12] P. Yiannacouras, J. G. Steffan, and J. 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*. New York, NY, USA: ACM, 2008, pp. 61–70.

[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*, aug. 2012, pp. 231 –236.

[14] C. Lee, "An algorithm for path connections and its applications," *Electronic Computers, IRE Transactions on*, vol. EC-10, no. 3, pp. 346–365, Sept 1961.