2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum

A Hierarchical Architectural Framework for Reconfigurable Logic Computing

Peng Li*, Angshuman Parashar†, Michael Pellauer†, Tao Wang*, and Joel Emer†‡
* Peking University † Intel Corporation ‡Massachusetts Institute of Technology
{peng.li, wangtao}@pku.edu.cn {angshuman.parashar, michael.i.pellauer, joel.emer}@intel.com

Abstract

Recently there has been growing interest in using Reconfigurable Logic (RL) for computation because of the significant performance gains that they can provide over traditional architectures on many classes of workloads. While there is a rich body of prior work proposing a variety of reconfigurable systems, we believe there hasn't been an attempt to clearly identify the architectural tradeoff spaces for an RL compute engine and to clearly separate architectural choices from implementation ones.

In this paper, we propose a taxonomy of architectural choices for RL computing. The taxonomy covers a multi-dimensional tradeoff space involving choices on operations, data types, states, sequencing, and communication primitives, and provides architects with a systematic framework for making decisions on these choices. We highlight the implementation and programmability consequences of such decisions, and wherever appropriate, punctuate the descriptions with examples of prior work that have made specific choices. Finally, we demonstrate how our proposed taxonomy is general enough to be hierarchically composed into a multi-level framework capturing the architectural design space of complex systems based on RL, such as heterogeneous systems comprising of traditional CPUs augmented with RL engines.

1. Introduction

The evolution of computer architecture in the past decade witnessed the saturation of single-thread performance scaling and the rapid rise of multi-core processors in an attempt to make use of the exponentially increasing transistors afforded by Moore's Law. Unfortunately, not all applications scale easily with increasing core counts. Consequently, some vendors have started integrating custom fixed-function logic blocks next to general-purpose processors to accelerate specific algorithms. Though accelerators could be designed with some limited flexibility in mind, their extensibility is still limited to what hardware designers envisioned at design time. This limits such architectures' agility in responding to emerging workloads. More importantly, architectures designed around arrays of fixed-function accelerators do not encourage programmer innovation to the extent that general-purpose processors have done in the past few decades of computing.

There is a middle ground between general purpose processors and full custom ASICs, namely, Reconfigurable Logic (RL) architectures such as FPGAs. RL architectures are composed of logic circuits that can be configured in the field to perform a variety of logic functions. While RL architectures are inherently less area and power efficient than fixed-function logic, they possess certain unique traits that allow them to efficiently execute a number of applications that are not amenable to multi-threading or vectorization. These traits include the ability to support workloads exhibiting immense fine-grained but irregular parallelism, the ability to perform custom bit-level manipulations, abundant configurable local storage structures and high-bandwidth on-chip networks to transfer data locally between pipeline stages at a fine granularity. Several applications have been shown to achieve tremendous speedups over CPU execution when ported to Reconfigurable Logic fabrics [1-3].

RL, particularly FPGAs, have traditionally been used either for prototyping ASICs or for replacing ASICs in low-volume deployments. An FPGA's LUT-based architecture allows for extremely fine-grained bit-level operations, which is useful for logic replacement. However, LUTs are often inefficient for expressing datapaths for algorithmic computation which usually involve coarser-granularity data operations. Coarser-grain RL architectures such as MATRIX[4] and PipeRench[5] have been proposed to address these inefficiencies.

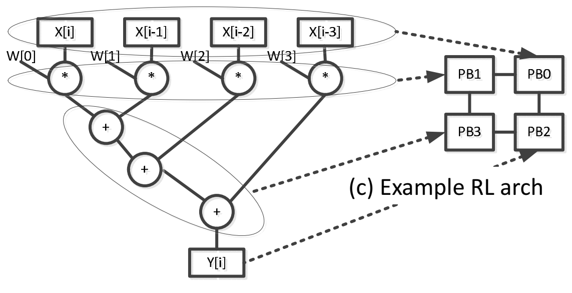
These are just a few examples in a rich body of literature on reconfigurable architectures. In defining each of these architectures, designers would have made decisions on a variety of architectural, microarchitectural and implementation choices. We believe there hasn't been an effort to systematically identify the architectural tradeoff spaces within which such decisions can be made, with a clear emphasis on separating *architectural* choices from microarchitectural or implementation ones. Examples of architectural choices for an RL engine include defining the state elements and operations supported by atomic *processing blocks* (PBs) in the RL fabric, the control model used to sequence operations within each PB and the inter-PB communication semantics.

In this paper, we attempt to establish a systematic architectural framework within which these choices can be made. The framework is in the form of a taxonomy of Reconfigurable Logic architectures, which we use to identify the tradeoff spaces, enumerate the architectural choices therein, and discuss the implications of these choices on programmability and implementation cost.

978-0-7695-4979-8/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPSW.2013.252

Y[i]=W[0]*X[i]+W[1]*X[i-1]+ W[2]*X[i-2]+W[3]*X[i-3]

(a) FIR Algorithm



(b) DFG of FIR

Figure 1. Typical Workload on an RL Architecture

2. Reconfigurable Logic Compute Architecture

The computation model of Reconfigurable Logic architecture is quite different from that on a traditional von-Neumann processor. To appreciate the range of architectural choices involved in defining such a computation model, consider the workload example presented in Figure 1, a Finite Impulse Response (FIR) filter. Attempting to construct an RL architecture that can efficiently execute this workload reveals a number of choices.

The first architectural choice is whether a PB contains internal state to store constant parameters (*W* in the example) or intermediate results.

The second choice involves the manner in which a PB gets its input data (*X* in this example) – does it have autonomous access to the system memory, or does an external agent (e.g. a CPU core) feed the required data to the PB?

The third choice involves the sequencing of different operations mapped onto a PB. If the four multiplications in this example are mapped onto a single PB, should each multiplication start based on a program counter in the PB, or a finite-state machine based on the arrival of data?

The fourth architectural choice is the semantics of the interconnection network used by different PBs to communicate with each other – What is the interconnection topology? Do they use channels or shared buffers to communicate?

The choices illustrated using this simple example in fact reflect fundamental architectural choices that define an RL architecture, namely, context-availability, memory-accessibility, sequencing model within a PB and the communication model between multiple PBs. The set of alternatives for each of these choices are shown in Figure 2. Defining an RL architecture involves selecting a set of these options. Each option naturally has implementation ramifications, as well as performance implications for workloads mapped onto the architecture. The example RL architecture shown in Figure 1(c) makes a set of choices that makes it an effective architecture to map our example

Local State [2.1]	With Internal Context		Without Internal Context
Remote State [2.2]	With Memory Access		Without Memory Access
Control [2.3]	PC Sequencing	FSM Sequencing	No Sequencing
Communication [2.4]	Non-Blocking Message Passing	Blocking Message Passing	Static Scheduling
	Clock-based Communication		Shared Memory

Figure 2. Architecture Taxonomy

FIR filter workload onto. The fabric has 4 Processing Blocks (PBs) connected with mesh interconnection is shown Figure 1(c). Communication between adjacent PBs in this example RL fabric takes a single cycle, which is much shorter than communication between multi-cores which typically takes thousands of cycles. All the load operations can be mapped onto PB0 since memory operation is probably to be serialized somewhere. The loaded data can be passed to PB1 for multiplication with constant W[j] saved in its local storage, and the result is pipelined to PB3 for the accumulation. The store operation of the final result is mapped onto PB2.

In the remainder of this section, we explore each of these architectural choices in detail and discuss tradeoffs associated with the possible options for each choice.

2.1 Local State

Programmer-visible state is typically pragmatically divided into small and fast local state and large and slow remote state. Local state is maintained in the set of architectural registers, and its remote state is the contents of main memory. It is reasonable to consider whether a PB contains programmer-visible local state or not. A PB without local state is simple to build and manage. If the PB is used for independent computations that do not require intermediate data to be maintained between executions, the need for maintaining local state inside the unit is obviated. However, if intermediate data does need to be transferred across computations (for example, the accumulated sum in an RL-based accumulator), a PB without local state has to return the intermediate data to an external agent (e.g. a CPU core or a global memory location), which needs to be sent back to the PB as a parameter for the next computation.

With support for local state, intermediate data can be saved inside the PB; this reduces data transfer cost in and out of the unit. For example, if a PB is being used for Regular Expression matching, having local state obviates the need to provide the regular expression for each chunk of the string being processed. The disadvantage of maintaining local state is that the state must be preserved (saved and restored) across context switches.

2.2 Remote State

The majority of persistent state that a software application maintains during its execution resides in main memory. A reconfigurable PB in the system may or may not be given the ability to directly access the main memory hierarchy. Without access to this remote state, input and output data for operations mapped onto this PB will have to be explicitly transferred to and from the unit by an agent such as a CPU core. This behavior is similar to that of a functional unit inside a CPU core, which relies on the core's load/store modules to access the system memory hierarchy. Many existing integrated RL systems follow this approach since it is easy to implement.

The alternative would be to allow the reconfigurable PB to access the system memory autonomously. The data access patterns from the RL could vary from application to application, which is why it could be beneficial to allow PBs in RL to handle them autonomously and independently of the CPU. Prior research on this issue has pointed out the importance of providing this capability to an RL[6]. A secondary advantage of memory access is that the address calculations can also take advantage of the RL fabric, and the local storage in the RL fabric can be exploited to buffer requests from memory independently from a processor.

2.3 Control

Internal control within each PB in a reconfigurable architecture can be achieved using a traditional von-Neumann model, where a Program Counter (PC) is used to index and read the instruction store and fetch a sequence of instructions for execution.

While this PC-based sequencing model is very general and effective at expressing single-threaded control, it is ill-suited to capture the more complex Finite State Machine (FSM)-like behavior of PBs in a reconfigurable architecture. In a PC-based architecture, determination of the next FSM state must be achieved either with a cascading sequence of if-else clauses or an indirect-branching mechanism – both of which are inefficient in a dynamic environment with fine-grained messages traversing along numerous input and output channels. However, the generality of this control model coupled with the depth of understanding of the model that architects have accumulated over decades makes it a viable choice for certain RL architectures.

The control model used by Configurable Logic Blocks (CLBs) on FPGAs stands in stark contrast to the sequential PC-based model. A CLB typically consists of a small number of Look-Up Tables (LUTs), registers and a set of configurable multiplexors (MUXes). Each operation of a CLB could be defined as the composition of a set of register reads, a set of input-wire reads, a set of LUT lookups, a set of register writes and a set of output-wire writes. Unlike a von-Neumann CPU, a CLB is first *configured* before usage, during which the contents of the LUTs and the configuration of the MUXes are set. These configurations are not altered during the execution of the algorithm. The entire CLB can, in a sense, be viewed as executing the same "operation" all the time, but the parameters of the operation depend on input data and local state. Thus, the sequencing model for a CLB is a trivial no-sequencing model where the same operation executes repeatedly.

It is possible to design medium-grain PBs that alleviate some of the limitations of the von-Neumann sequencing model described earlier by drawing inspiration from CLBs. The key idea is to separate data computation from control (next state) computation based on the intuition that architectural structures that are optimal for data transformations may be distinct from structures that are optimal for control computation.

Consider a PB with a set of operations stored in an instruction memory, but for which the next state is not set by an instruction but is evaluated using a Finite-State Machine (FSM) controlled by distinct set of control-evaluation operations exposed to the programmer by the architecture. The inputs for these operations are a subset of the state of the PB – condition registers, ALU condition codes/flags, bit extractions from data registers, bits from input wires arriving at the PB, miscellaneous status bits, etc. The programmer is allowed to express control computations with an architecturally-defined set of operations. An example for this FSM-sequencing could be a switch processor that decides which instruction to execute based on a function of a subset of header bits in incoming data packets. These functions could be evaluated using hardware optimized for such computations instead of using the datapath. This kind of architectural separation of control and data paths results in far more efficient PBs than traditional PC-based sequencing.

2.4 Communication and Interconnection

How PBs communicate and coordinate with each other to execute a complete workload is a major decision in RL architectures. In this section we cover three main approaches: clock-based communication, message passing (blocking or non-blocking), and shared memory.

2.4.1 Clock-based communication

The first approach, used classically by FPGAs, is to use physical clocks to coordinate communication between PBs. In this scenario control of all communicating PBs and the interconnection network between them is coordinated by clock edges. All PBs use a clock edge as a signal to read data in, operate on that data, and produce new data out. Because of this, the maximum clock rate of the entire system is determined by the rate-limiting step in the communication graph of the workload. Devices may provide separate physical clock domains so that the limiting effect can be minimized.

2.4.2 Message Passing

While clock-based communication is a good choice for logic replacement of ASIC, direct message passing between multiple PBs is desirable for RL fabrics used for computation because of its high efficiency. Synchronization between the sender and receiver PBs during message passing can have multiple choices depending on application characteristics and the underlying physical interconnection network.

While interconnection network topology could have innumerable options, whether it is latency sensitive [7] or not is important for message passing implementation. In a latency-sensitive network, the latency from any source to destination is statically deterministic. One example would be if each PB is only connected with its nearest-neighbor PBs with fixed latency. In a latency-insensitive network, PBs can be connected via an on-chip network (OCN) that directs messages (often divided into packets, and possibly further into flits). Latency through the network is a function of the dynamic load on shared network resources. This dynamic variability means that PBs cannot make synchronous assumptions about the amount of time that data will take to transfer from producer to consumer. Therefore, communication cannot be statically scheduled onto a latency-insensitive network at compile time. Instead, non-blocking or blocking communication mechanism can be used on latency-insensitive network with dynamic network delay.

Active-polling and interrupt are the most commonly used non-blocking communication. Using active-polling, the sender PB will use dedicated instructions to periodically check for the availability of the channel (e.g. through a busy loop) and send the message at earliest available time slot, while the receiver PB will use also dedicated instructions to periodically check for the arrival of the message. The PB FSMs must be given access to the empty/full state of the network interface FIFOs to detect when production or consumption is allowed. This can degrade efficiency for sequential PC-based PBs since active-polling loops which repeatedly issue instructions to query FIFO status are power-inefficient.

Another option for non-blocking communication is to use interrupts. The sender/receiver PB is interrupted from its current task upon the availability of the channel or a message. At this point, the PB enters an interrupt service routine to finish the communication action before returning to its original task. Interrupt-handling usually consumes additional energy to switch back-and-forth between the main thread and the interrupt service routine.

For frequent communications, a blocking mechanism can be used, where the sender/receiver PB would be blocked on the producing/consuming instruction until that message is actually sent/received. Typically, blocking communication is used on some latency-insensitive interconnection with FIFOs to buffer the messages. Blocking communication can be more energy-efficient for frequent message-passing if the waiting PB avoids expensive stages like instruction fetching and decoding.

The control flows of certain compute-intensive applications (e.g. some signal processing applications) are fairly simple. For such applications, a designer can (at least theoretically) statically schedule and map the application onto a latency-sensitive network carefully and efficiently. A well-designed compiler can also statically schedule, place and route the Control/Data Flow Graph onto the underlying latency-sensitive interconnection using sophisticated algorithms[8]. Through static scheduling, these compute-intensive applications can be mapped onto target architectures efficiently with minimal bubbles, leading to high performance and energy efficiency.

2.4.3 Shared Memory

Although direct message passing is desirable for its high-performance and low-power, there still might be a need for shared memory communication between PBs. The problem is to what extent it is supported. At one extreme, an architecture may only allow a few PBs to access memory at all. (Perhaps a pipeline is created between loading PBs, operating PBs, and storing PBs). Less restrictive is to statically partition memory regions between PBs. This allows all PBs to perform memory operations, but only on disjoint sections suitable for scratchpads. At the other extreme, if every PB can read and write the same memory region independently, then some sort of coherence mechanism is required to remove harmful data races. As with a cache coherence protocol, this can result in traffic in an on-chip network similar to the traffic generated by direct PB-to-PB messaging.

3. Hierarchical Architecture Framework

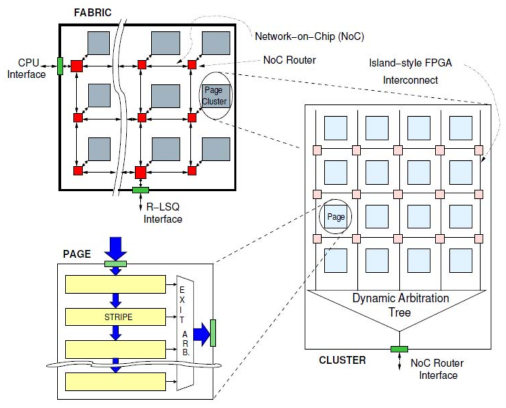
The set of architectural choices presented in Section 2 can be composed into a multi-level hierarchical framework that can be used to define complex RL systems, including hybrid CPU/RL systems, with a different set of architectural choices at each level. In fact, many existing RL (and hybrid CPU/RL) architectures can be mapped and studied using this framework.

Figure 3(a) shows Tartan[9], an existing hierarchical RL fabric that we use as an illustrative example. The top level of the system consists of a CPU core (not shown in the figure) and a Reconfigurable Fabric (RF). They use non-blocking communication (both active-polling and interrupts) and shared memory to communicate with each other. The RF consists of multiple homogenous Page Clusters connected by a latency-insensitive dynamic-routed packet-switched on-chip network. Blocking message passing is used between Page Clusters where communication is sporadic and unpredictable. A Page Cluster consists of multiple homogenous Pages connected by island-style FPGA interconnection while a Page is made up of multiple Strips connected by a partial crossbar. A Strip is made up of multiple Processing Elements (PEs, not shown in the figure). Each PE has its own register file and ALU configured statically. It can access the memory directly with very limited bandwidth. Communication between Pages/Strips/PEs is statically scheduled message passing with lower low-latency and energy consumption.

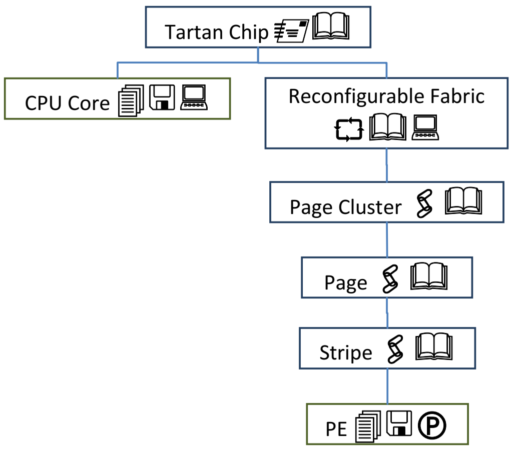
Figure 3(b) shows a visualization of the Tartan architecture using our hierarchical architectural framework. Figure 3(c) explains the icons we have chosen for the decisions from Figure 2.

The definition of control for a non-leaf PB in the hierarchy isn’t straightforward. A non-leaf PB will be configured to accelerate a specific task before its execution. This reconfiguration and execution process is similar to the instruction fetching and execution in a traditional processor. Instructions in a non-leaf PB are encoded with instruction sequences of all child-PBs and are much longer than typical CPU instructions. The execution time is also much longer and typically non-deterministic in advance, somewhat like (very complex) micro-coded CISC instructions. In Tartan, the entire RF fetches and executes a new instruction (entire configuration) when switching to a new application. Another example for sequencing a non-leaf PB is Tabula[10], where the whole reconfigurable array can sequence rapidly between different configurations each cycle.

Using the proposed architecture framework, we visualize 6 other typical reconfigurable systems shown in Figure 4.



(a) Tartan Architecture. Source: [9].

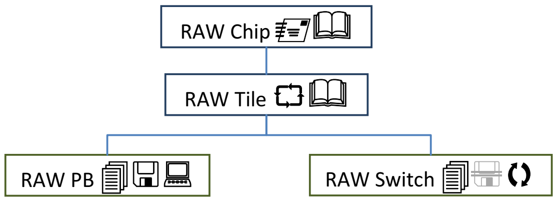


(b) Description for Tartan Architecture using Proposed Framework

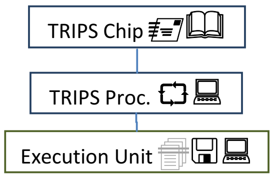
With Internal Context 🗐	Without Internal Context
With Memory Access 💾	Without Memory Access
PC Sequencing 💻	FSM Sequencing
No Sequencing Ⓟ	Non-Blocking Message Passing
Blocking Message Passing	Static Scheduling
Clock-based Communication 🕒	Shared Memory 📖

(c) The Meaning of Icons in Architecture Description

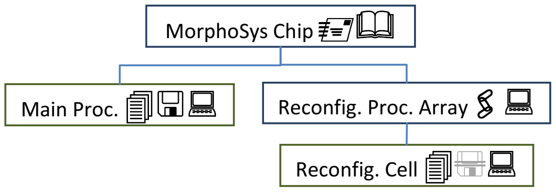
Figure 3. Tartan: An Example for Hierarchical RL Architecture and its Architecture Description



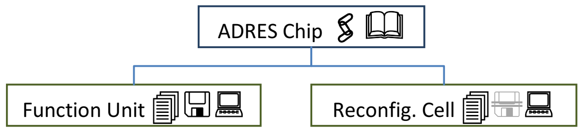
(a) MIT RAW[11]



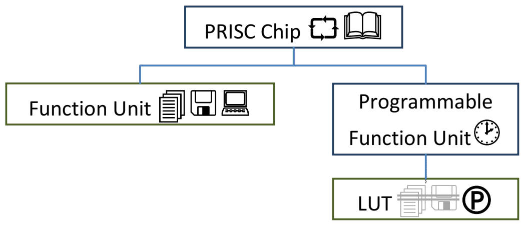
(b) TRIPS[12]



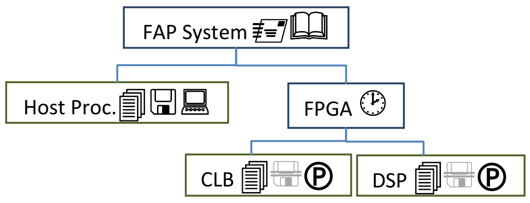
(c) MorphoSys[13]



(d) ADRES[14]



(e) PRISC[15]



(f) FAP [16]

Figure 4. Typical RL System Description using Proposed Architecture Framework

4. Concluding Remarks

Reconfigurable Logic fabrics show tremendous potential in serving as compute platforms for a variety of applications. While there has been a significant amount of prior research quantitatively establishing the performance potential of RL, this paper attempted to take a first step towards proposing a hierarchical architectural framework for describing and understanding the multiple dimensions of architectural decisions that need to be made while designing an effective CPU/RL platform. Our framework classifies architectures by availability of local and remote state, PB sequencing, communication schemes between multiple PBs, and access to shared memory. Furthermore, these choices are expanded into a tree-based hierarchy which gives insight into complex hierarchical RL systems. We believe this insight will aid architects in conveying their ideas to low-level programmers and compilers for efficient workload mapping.

References

[1] M. D. Galanis, G. Dimitroulakos, and C. E. Goutis, "Speedups and Energy Reductions From Mapping DSP Applications on an Embedded Reconfigurable System," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on,* vol. 15, pp. 1362-1366, 2007.

[2] G. Zhang, P. Leong, C. Ho, K. Tsoi, C. Cheung, D.-U. Lee, R. Cheung, and W. Luk, "Reconfigurable acceleration for monte carlo based financial simulation," in *IEEE International Conference on Field-Programmable Technology*, 2005, pp. 215-222.

[3] C. Liang and X. Huang, "Mapping Parallel FFT Algorithm onto SmartCell Coarse-Grained Reconfigurable Architecture," in *Proceedings of the 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2009, pp. 231-234.

[4] E. Mirsky and A. DeHon, "MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1996.

[5] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler," *IEEE Computer,* vol. 33, pp. 70-77, 2000.

[6] J. Carrillo and P. Chow, "The effect of reconfigurable units in superscalar processors," in *Proceedings of the International Symposium on Field Programmable Gate Arrays*, 2001, pp. 141-150.

[7] K. E. Fleming, M. Adler, M. Pellauer, A. Parashar, A. Mithal, and J. Emer, "Leveraging latency-insensitivity to ease multiple FPGA design," in *FPGA '12 Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pp. 175-184.

[8] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," in *Design, Automation and Test in Europe Conference and Exhibition*, 2003, pp. 296-301.

[9] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, M. Budiu, and S. C. Goldstein, "Tartan: Evaluating Spatial Computation For Whole Program Execution," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006, pp. 163-174.

[10] T. R. Halfhil, "Tabula's Time machine Rapidly Reconfigurable Chips Will Challenge Conventional FPGAs," *Microprocessor Report,* 2010.

[11] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to Software: Raw Machines," *IEEE Computer,* pp. 86-93, 1997.

[12] K. Sankaralingam, R. Nagarajan, H. Liu, J. Huh, C. K. Kim, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ILP, TLP, and DLP Using Polymorphism in the TRIPS Architecture," in *30th Annual International Symposium on Computer Architecture (ISCA)*, 2003, pp. 422-433.

[13] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "MorphoSys: An Integrated Reconfigurable System for Data-Parallel Computation-Intensive Applications," *IEEE Transactions on Computers,* vol. 2000, pp. 465 - 481, 2000.

[14] B. Mei, S. Vernalde, D. Verkest, H. D. Man, and R. Lauwereins, "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix," in *13th International Conference Field Programmable Logic and Application*, 2003, pp. 61-70.

[15] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware programmable functional units," in *Proceedings of the 27th annual international symposium on Microarchitecture*, 1994, pp. 172 - 180.

[16] L. Liu, N. Oliver, C. Bhushan, Q. Wang, A. Chen, W.Shen, Z. Yu, A. Sheiman, I.McCallum, J. Grecco, H. Mitchel, D. Liu, and P. Gupta, "High-performance, energy-efficient platforms using in-socket fpga accelerators," in *FPGA' 09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, 2009, pp. 261-264.