Vision for Liquid Architecture

Roger D. Chamberlain
Ron K. Cytron
Jason E. Fritts
John W. Lockwood

Roger D. Chamberlain, Ron K. Cytron, Jason E. Fritts, and John W. Lockwood, "Vision for Liquid Architecture," in *Proc. of Next Generation Software Workshop,* April 2006.

Dept. of Computer Science and Engineering
Washington University

Dept. of Mathematics and Computer Science
Saint Louis University

Vision for Liquid Architecture

Roger D. Chamberlain[1], Ron K. Cytron[1], Jason E. Fritts[2], and John W. Lockwood[1]

[1]Washington University
Dept. of Computer Science and Engineering
Saint Louis, MO 63130 USA
{cytron,roger,lockwood}@wustl.edu

[2]Saint Louis University
Dept. of Mathematics and Computer Science
Saint Louis, MO 63103 USA
jfritts@slu.edu

Abstract

In the liquid architecture project, we are exploring ways in which architectural flexibility can be exploited to improve the execution properties of individual applications. Here, we report on successes we have had to date in this area, and present our vision of where this research should proceed into the future.

1. Introduction

Traditional general-purpose processors have physical properties and characteristics that are fixed in many cases well before the application is designed. The hardware design team has completed its work, the processor has been fabricated, and the software design team (with the help of compilation tools) must conform their applications to the limitations and constraints of the pre-existing hardware. The instruction set architecture (ISA) forms a stable interface between the hardware and software design teams, yet this interface hides many details of the system's microarchitecture; those details may dramatically influence the application's predictability and performance.

The motivation for the above state of affairs is strongly influenced by the high design and fabrication costs of general-purpose processors. Literally billions of dollars are spent on fabrication lines, and hundreds to thousands of man-years are expended on the part of the hardware design team in an attempt to ensure that "typical" applications (whatever that means) achieve the best performance that can be managed. Worse still, processors are often marketed based on performance statistics on popular benchmarks that may have little in common with any given application.

Reconfigurable hardware, in the form of Field Programmable Gate Arrays (FPGAs), offers a number of alternatives. First, when deploying soft-core processors on an FPGA, microarchitecture implementation and configuration can be quite flexible. Second, FPGAs enable the integration of both general-purpose processors and application-specific functional units or coprocessors on the same chip. Third, the hardware design need not be fixed prior to software development. Hardware and software can be designed in tandem (i.e., hardware/software codesign). In short, significant opportunities for customization are available, providing the ability to improve performance, predictability, reliability, size, power consumption, or any other property of interest.

A number of these properties are of particular interest to embedded applications, where the design constraints are often much tighter than for general-purpose systems (i.e., those on the desktop or in the server room). The ability to alter the physical properties of computational engine provides significant benefits for embedded applications.

Our group has been exploring the potential benefits achievable with a *liquid architecture*, in which the properties of the execution platform can be more responsive to the needs of the application. Instead of the software having to completely conform to the reality of a fixed hardware platform, in a liquid architecture the hardware also conforms to the needs (requirements) of the application. In this paper, we briefly describe our past work on the liquid architecture project and present our vision of the future for research into flexible architectures.

This work was supported by NSF grant CNS-0313203.

2. Current Work

The focus of the *liquid architecture* research is investigation into techniques by which the architecture of the underlying processor can adapt to the specific needs of individual applications. Our experimental work uses the Field-programmable Port Extender (FPX) platform [15]. The FPX, built at Washington University, provides an environment where FPGA designs can be interfaced with memory and a high-speed network interface. We have deployed the LEON [14] (a SPARC V8 compatible, soft-core processor), a memory controller, control processor, and statistics module (described below) on the FPX. Custom, application-specific functional modules are interfaced either through the high-speed bus (AHB) or the standard SPARC coprocessor interface. OS support includes both uClinux [25] when the memory management unit (MMU) is absent and Linux kernel 2.6.x when the MMU is present. The liquid architecture system is illustrated in Figures 1 and 2 [12, 19, 21].

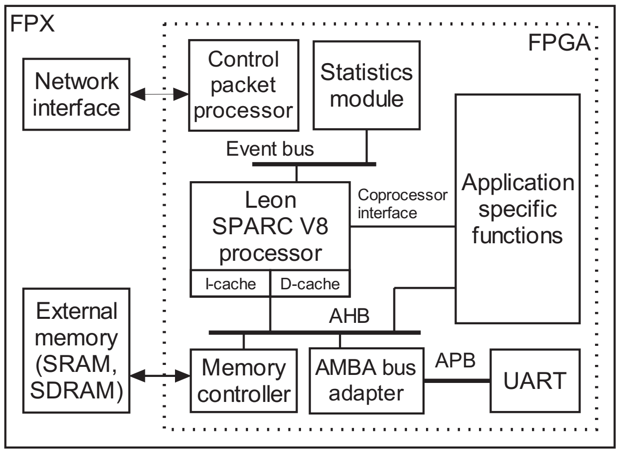

Figure 1. Liquid architecture block diagram.

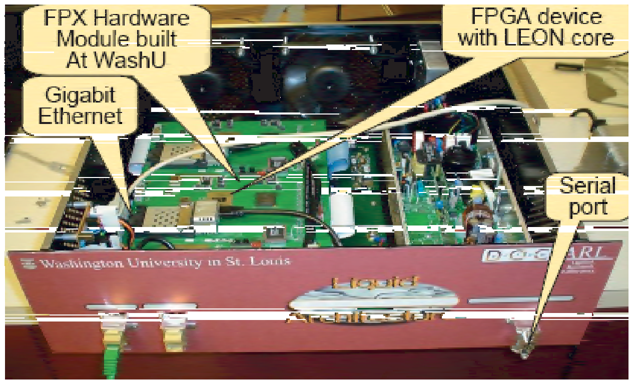
FPX Hardware Module built At WashU
FPGA device with LEON core
Gigabit Ethernet
Serial port
Washington University in St. Louis
Liquid Architecture

Figure 2. Liquid architecture photograph.

To date, our research activities in the liquid architecture group include the following:

- We have developed a statistics module that is designed for non-intrusive monitoring and reporting of application performance [10]. The statistics module is constructed using dedicated monitoring logic on the FPGA that can be dynamically allocated at execution time to measure virtually any signal of interest within the system. In addition, monitoring can be selectively enabled/disabled on method boundaries. The statistics module has recently been enhanced to work seamlessly with the MMU and the Linux 2.6.x OS.

- We have implemented a technique for microarchitecture configuration tailored to a particular application using a binary integer programming formulation [17, 18]. Rather than rely on simulated performance estimates, this technique uses actual measured execution times to evaluate application run time.

- We have investigated the utility of a specialized cache subsystem aimed at reducing unnecessary writebacks [9]. Called a *dusty cache*, it avoids flushing of cache blocks that have been altered and then returned to their original value. This is a common circumstance when reference counting for garbage collection. It is currently being enhanced to avoid writebacks of unallocated stack frames.

- We have extensively used a common biosequence search application for benchmarking and hardware/software codesign experience [13]. The BLAST nucleotide search engine has been accelerated to provide greater than a 30× speedup over a high-end Pentium workstation.

- Analytic performance models for pipelined applications have been developed that account for variability in execution requirements at individual pipeline stages and also evaluate the blocking probabilities associated with fixed-length, stage-to-stage queues.

- We are investigating the use of partial reconfiguration of the FPGA to dynamically load application-specific functional units at execution time.

In addition to the research activities described above, we are now using the liquid architecture platform as the core of our senior-level hardware/software codesign course [4].

3. Vision for the Future

Having described what we have been doing in the previous section, here we articulate our vision for future research in this area.

3.1. Cutting across Interfaces

Computing systems designed with hardware and software components typically involve standard interfaces at various levels. The ISA is but one example, a coprocessor interface is another. These interfaces are widely adopted to simplify the development tasks and to provide a level of abstraction to promote interchangeable and reuseable components. While the interfaces facilitate system design and implementation, they carry a number of disadvantages:

- Traversal of the interface can be a source of overhead. Indeed, consideration of what can be deployed on either side of the interface is typically limited by the overhead of using the interface. For example, the nature of what might be deployed in a coprocessor is dictated largely by the cost of activating the coprocessor, sending it the data necessary for its function, and retrieving the values produced by the coprocessor.

  While the standard interfaces should be kept as a development and implementation mechanism, reconfigurable platforms offer the opportunity to optimize across and through the interfaces to obtain better performance and predictability. We give an example of this kind of optimization below.

- The structures hidden by an interface may be useful for direct access by functions deployed across the interface. The register files, cache subsystems, and function units of a processor may be useful for associated firmware or coprocessor logic, but the ISA does not expose those components directly.

  The resources of a soft-core processor could be made available to all logic involved in a computation, so as to improve overall performance and increase resource utilization. In our vision, this is a form of a code-generation or synthesis problem whose solution could push software/hardware codesign to new levels for reconfigurable devices.

- Some interfaces, such as JNI and Beans (component models), are currently deployed only in software, but these interfaces also make sense for hardware/software codesign. For example, activation of hardware logic via JNI simplifies the software programming task because of the already understood semantics of JNI method calls. Logic activated in this manner may require obtaining values from the runtime heap—activities also supported by JNI but currently unavailable via any standard interface to hardware.

  We are currently pioneering a Java Firmware Interface, akin to the Java Native Interface, that would allow firmware to interface with Java structures available through JNI, such as the runtime heap and method activations.

Formal interfaces play a central role in application development. By articulating a formal interface, development on both sides of that interface can proceed without undue attention to the implementation details on the other side. Applications developed in a high-level programming language are generally unconcerned with the ISA of the target architecture, given a compiler that acts as an appropriate interface to the ISA. Similarly, the compiler writer sees the ISA, but is generally unconcerned with microarchitecture implementation, trusting the hardware designers to do their best job on that side of the ISA interface.

From a *functional* view of an application, details across such interfaces can remain abstract: correct results will be provided if the interfaces are correctly defined and consistently used. However, for *nonfunctional* properties—performance, predictability, power—the otherwise hidden details can be of paramount importance. In previous work, we have seen the value of automatically configuring microarchitectures on a per-application basis [17, 19]; extensions to that work are considered in Section 3.3. For many embedded-systems applications, these nonfunctional properties loom large in terms of requirements placed on the end system; research beyond automatic configuration is needed to customize an implementation across formal interfaces.

For example, consider a sensor whose data must be processed at some rate r. When the sensor is active, a high-speed commodity processor is necessary to pace the sensor's ingest rate; when the sensor is inactive, no processing is necessary. On a commodity processor, a high clock rate is necessary to pace the sensor's ingest rate. On the other hand, specialized logic could be deployed to process the sensor data, operating at a much lower clock rate and consuming correspondingly less power.

As another example, consider applications deployed as coprocessors on platforms like MIPS and SPARC. The coprocessor interface serves as a standard that allows processor and microarchitecture development to occur independently of the coprocessor implementation. As a result, the processor and coprocessor share only the most common structures that can be defined for them—typically, only a small set of registers and control signals—and the vast resources of a given MIPS or SPARC implementation are hidden from the coprocessor. For standardization purposes, and for ASIC processor implementations, such abstraction is necessary. However, for soft-core processors used in the applications we consider, the coprocessor interface is useful from a development standpoint, but it stands in the way of high performance.

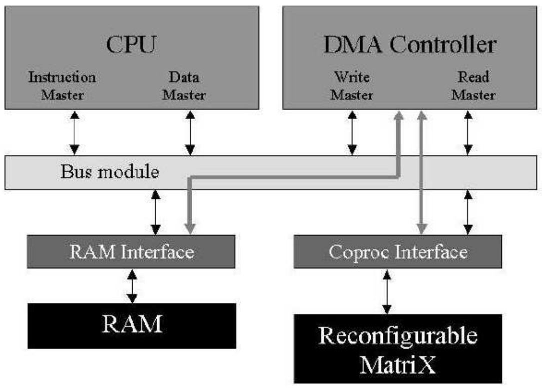

Figure 3. Application architecture from [1].

For example, consider Figure 3, which shows a signal-processing (wavelet) application architecture [1] involving a coprocessor. The coprocessor is forced to obtain data (via DMA) from RAM. That same data was streamed through the CPU and could have been presented to the coprocessor from the CPU's cache, but the cache is not part of the formal interface between the processor and coprocessor. With the coprocessor interface, the data must be funneled through a narrow set of registers or slow RAM to reach the coprocessor.

While the above is the best that can be done with an ASIC CPU and the formal coprocessor interface, much more can be done for soft-core CPUs and reconfigurable platforms: By cutting across the interface, the coprocessor could have direct access to the on-chip cache, dedicated multipliers, and other resources offered by the soft-core implementation.

We believe that compile-time analysis and transformation techniques are needed, that would allow a system implemented with standard interfaces to be optimized through the interfaces to make better use of resources on both sides. To demonstrate feasibility of our idea, consider the example above and its optimization across the coprocessor interface. Using analysis akin to *method inlining*, the VHDL code of the soft-core processor and coprocessor implementation can be analyzed, so that the true source and sink of the data across the interface can be more readily determined. Accesses to the coprocessor registers on the coprocessor side are then automatically transformed into on-chip RAM accesses, with the coprocessor code woven into the soft-core implementation.

Of course, this transformation cannot happen on ASIC implementations, but for the reconfigurable platforms we investigate, such transformation is easily achieved, even on a per-application basis.

3.2. Robust Hardware/Sofware Codesign

Because of its popularity, its widespread and standardized implementations, and its ability to interface with other systems and languages, Java is a reasonable choice for experimentation in the hardware/software codesign space. But how should Java interface with hardware components? We are currently considering the following ideas:

JFI or Java Firmware Interface: Java already has an interface for other programming languages, known as JNI for Java Native Interface. JNI offers mechanisms for activating methods on both sides of the interface and for reading and writing Java (heap-allocated) data. We seek to implement those same methods for firmware, so that construction of the Java program to activate, inteface with, and retrieve results from hardware-deployed functionality is as simple as writing JNI code today. The semantics of this mechanism, as for JNI, is synchronous, in that the Java thread invoking JFI methods would block until the method completed.

Beans Because much of hardware logic is meant to operate asynchronously, we are also considering a Beans-based implementation of Java hardware components. Our work here is inspired by [8], who have considered deploying hardware components in this fashion.

Because of our interest in optimizing across interfaces, each of the above codesign environements is only the beginning of the solution. For a soft-core architecture, we can consider a view from the Java application, through JFI, and into the logic deployed on the firmware side. Optimization across those interfaces can improve the nonfunctional properties we describe above.

As examples of this kind of effort, we are currently involved with Boeing and the University of Dayton in providing a compiled Java environment [7] that interfaces with a firmware storage-management subsystem and with firmware logic to accelerate certain computations. The interfaces are managed manually at present, but their automation would bring this kind of programming and experimentation within reach of a much wider audience.

3.3. Microarchitecture Autoconfiguration

While interfaces abstract detail and thus simplify component interaction, the structures hidden by the interface may be critical for an application's performance. For example, an ISA abstracts cache subsystems, branch prediction, and out-of-order execution, yet those microarchitecture structures can significantly impact application performance.

Soft-core processors offer the opportunity to change their microarchitecture support quickly, in response to the needs of a given application. Our experience shows that application-specific configuration of the microarchitecture can play an important role in an application's performance.

Reconfigurable architectures based on FPGAs offer significant advantages for embedded applications, not only in the versatility of the dynamically reconfigurable architecture, but also in the ability to perform non-intrusive, cycle-accurate hardware profiling, i.e. cycle-accurate performance evaluation can be performed in real-time for free. Conversely, the flexibility of an FPGA-based, dynamically-reconfigurable architecture provides an enormous design space for hardware and soft-core based computer architectures, so effectively exploring that design space presents a difficult and potentially very time-consuming problem.

The classical performance analysis tool used to explore the architectural design space is simulation. However, simulation generally is plagued both by long execution time and uncertainty as to the fidelity of the underlying model. We avoid these issues by exploiting the reconfigurable nature of FPGAs and use direct execution as our performance measurement tool of choice. By adding explicit monitoring and reporting logic to the system, we can achieve visibility into any component or subsystem we desire, all with zero impact on the performance of the monitored system [10]

To effectively explore the design space and determine a highly-effective architecture for an application or set of applications, we envision a 4-stage methodology to maximize the exploration of this design space in the most efficient fashion. This 4-stage methodology is designed to quickly find the architecture designs that most efficiently support the individual stages of an application. It uses phase detection to identify the unique phases in an application, a statistically-rigorous evaluation methodology [26] to expeditiously find the most important architecture parameters for each phase of an application, and a binary integer programming method we developed [17] to select the appropriate parameters based on their importance and the resource constraints of the FPGA. The methodology is intended to identify the best hardware parameterizations for each phase of an application in both a fast and statistically-rigorous fashion, leveraging the real-time full-system hardware evaluation capabilities of FPGAs to effectively address the challenges of computer architecture evaluation [23].

The first stage of the methodology uses phase detection, as proposed by Cook et al. [6] and Sherwood et al. [22], to detect the unique phases within a program. Phase detection is now starting to be used as a tool for fast software-based simulation, such as Srinivasan et al.'s method [24] that only simulates each unique phase of a program once. However, we envision using it in an alternative fashion. Since FPGA-based architectures are dynamically reconfigurable, the architecture can be dynamically adjusted for unique phases of a program, and we intend to use phase detection to enable design space exploration that considers alternate architectures for separate phases in the application.

The second stage of the methodology uses Yi et al.'s [26] statistically-rigorous approach to performance analysis. Traditional regression-based design space exploration methods require kN simulations/evaluations in order to do single-parameter evaluations, where N is the number of architecture parameters and k is the average number of values tested in each regression of a single parameter. Even worse, evaluation of two-parameter interactions for all parameters requires an exponential number, k^N, of evaluations. Conversely, Yi et al.'s method uses the Plackett and Burman (PB) statistical method [20] to rank the importance of each parameter for both single parameters and two-parameter interactions using only about $2N$ simulations/evaluations. Furthermore, it does so in a statistically-rigorous fashion, supporting the evaluation recommendations put forth by Skadron et al. [23]. Yi et al.'s method was originally proposed for high-performance general-purpose processor research, but the theory is applicable to reconfigurable architectures as well. In using their method for reconfigurable architectures, not only can we consider the large number of parameterizations possible in a soft-core processor, but we can also use the PB method for each phase of a program (with only one run needed to cover all phases of the program), and thereby identify the most important parameters for each phase of an application. Hence, this method will quickly rank the importance of each architecture parameter/coprocessor for each phase of an application.

After identifying the important single parameters and two-parameter interactions for each phase of an application, in the third stage of the method we perform standard regressions (including generating covariance matrices from multivariate regressions for important two-parameter interactions) in order to quantify the effects on execution time, area, power, reliability, etc. for these parameters and coprocessors.

In the final stage of the methodology, we use our recently developed binary integer programming technique [17] to select microarchitecture parameters for maximizing performance given constrained resources. This method is both feasible and scalable, so can readily support a large number of architecture parameters. While initial use of this method was limited by its reliance on application execution time as the single performance metric, we now have the capability to use arbitrary metrics on constituent subsystems (e.g., cache hit and miss rates) which will allow us to more fully understand the potential interactions between architecture parameter settings. A clear advantage of this will be the ability to reason about parameter interations in a more rigorous way, making definitive judgements rather than looking for correlations in empirical performance data.

3.4. Performance Predictability

While general-purpose computing systems are typically optimized for best average-case performance, in embedded systems it is often the the worst-case performance that drives design decisions. Real-time systems include performance in their correctness criteria, and the penalties for not succeeding (i.e., missing a deadline) can range from minor inconvenience (for some soft real-time requirements) to total system failure (for some hard real-time requirements). As a result, the *predictability* of system performance is a critical factor in the design process, and the use of reconfigurable hardware in the design of these systems provides a number of opportunities to improve the performance predictability of the resulting deployed application. For example, instantiation of an object in firmware makes the instance variables available in firmware registers—always one cycle away—as compared with the memory subsystem which could take from one to hundreds of cycles to reach the variable.

Intrinsic in the ability to ensure a given performance level is the need to understand the performance implications of design decisions that are being made. This understanding is getting more difficult, as systems become more complex and hierarchical. Abstraction barriers that are introduced to simplify the design process when one is concerned primarily with average-case performance end up making the design task much more difficult when one is concerned with meeting real-time constraints. Examples include both hardware abstractions (e.g., the ISA hiding microarchitectural details such as cache hierarchies) and software abstractions (e.g., the unintended priority inversion that happens in the OS when a low-priority interrupt is allocated processor resources instead of a high-priority task). Research has been undertaken to address the need for predictability in a number of areas, including cache subsystems [5, 11], branch prediction [2], instruction issue in SMTs [3], and open-source operating systems [16].

There are many opportunities to improve performance predictability by exploiting the unique capabilities that exist in reconfigurable systems. A simple example is the fact that a set of functionality that is deployed in dedicated logic typically has end-to-end performance characteristics that are well understood at design time. Once inputs have been provided, some known number of clock cycles later the results are available. A second example is the uncertainty generally associated with a cache memory hierarchy. Customized memory subsystems can be designed for predictable performance rather than best average-case performance.

An area in which we see immediate opportunity is the management of real-time structures for scheduling, thread management, synchronization, and priority management and inheritance. Programs are currently at the mercy of an operating system to provide those functions predictably and reliably. Unfortunately, the latest releases of stable, open-source systems do not provide such functions. Patches to those systems to obtain the desired properties are fragile; custom, real-time operating systems (such as VxWorks) can be difficult and expensive to use.

As an example, consider the access to a semaphore that is enforcing mutual exclusion for a critical section of code. If the access to the semaphore is via a multi-master system bus (e.g., one with two or more processors or smart peripherals), there can be significant uncertainty in the delay introduced by the bus itself. In a system with reconfigurable logic, it is straightforward to deploy a multiport hardware semaphore that is accessed completely independently of the system bus. Access by the processor's ISA can still be via common load/store mechanisms; however, the physical path no longer traverses the bus and therefore avoids the performance uncertainty introduced by the bus.

Note that the above example illustrates not only the ideas here in this section but also those of Section 3.1 above. A classical resource, a semaphore, is referenced by the application in the traditional way. However, the physical deployment of the resource has been altered to improve one or more aspects of the application's execution (in this case improving performance predictability).

4. Conclusions

When we began our research on liquid architectures, reconfigurable platforms were insufficiently mature to host real applications. While FPGA chips were available, they were not typically deployed with the accessories necessary to offer them access to high-speed networks or reasonable amounts of memory. They were considered slow, small, and difficult to use.

We developed and constructed liquid platforms with high-speed access to networks and we developed tool-chain support to enable reprogramming the platforms from any Internet connection. That infrastructure in turn supported several new directions of research that would not have been possible without reconfigurability.

- Application-specific microarchitecture configuration allows a soft-core processor implementation to be tuned with respect to performance, area, or power. Such a problem cannot be considered for an ASIC processor implementation, and the scope of experimentation necessary to solve the optimization problem required the tool-chain infrastructure we developed.

- Performance profiling that is accurate and completely nonintrusive has long been a dream of application developers. We have developed and demonstrated the performance of an efficient performance-monitoring circuit that would not have been possible without reconfigurability.

- Almost every application relies on efficiency of the storage subsystem. While caches generally serve an application well, we identified an important area in which standard cache structures do not provide the best performance. We developed the idea of a "dusty" cache [9] and showed that its performance on reference-counting garbage collectors is superior to standard caches. This kind of experimentation was greatly facilitated by the liquid architecture system. We were able to implement the cache structure and obtain measurements of its performance at full-speed, without recourse to simulation or modeling, which can give only approximate characterizations of behavior.

During the course of our work, reconfigurable platforms matured in the manner we predicted. Today one can buy affordable FPGA-based systems that offer most of the amenities found on high-end systems. Standard versions of Linux run on these boards, and they have the memory and network capabilities that make them viable as platforms for Next-Generation Systems. The boards are manufactured in quantity by companies that are likely to stay in business.

From a codesign perspective, the advent of the affordable FPGA-based system is a strong motivation for the research problems we have identified in this paper. To use these systems as if they were only smaller desktop machines would make no use of their reconfigurability and belie their true potential.

References

[1] S. Bilavarn, E. Debes, P. Vandergheynst, and J. Diguet. Processor enhancements for media streaming applications. *Journal of VLSI Signal Processing-Systems for Signal, Image, and Video Technology*, 41(2), September 2005.

[2] F. Bodin and I. Puaut. A WCET-oriented static branch prediction scheme for real time systems. In *Proc. of 17th Euromicro Conf. on Real-Time Systems*, July 2005.

[3] F. J. Cazorla, P. M. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. Predictable performance in SMT processors. In *Proc. of Conf. on Computing Frontiers*, pages 433–443, Apr. 2004.

[4] R. Chamberlain, J. Lockwood, S. Gayen, R. Hough, and P. Jones. Use of a soft-core processor in a hardware/software codesign laboratory. In *Proc. of Int'l Conf. on Microelectronic Systems Education*, June 2004.

[5] D. Chiou, P. Jain, L. Rudolph, and S. Devadas. Application-specific memory management for embedded systems using software-controlled caches. In *Proc. of Design Automation Conf.*, 2000.

[6] J. Cook, R. L. Oliver, and E. E. Johnson. Examining performance differences in workload execution phases. In *Proc. of the 4th IEEE Int'l Workshop on Workload Characterization*, December 2001.

[7] A. Corsaro and R. K. Cytron. Implementing and optimizing real-time Java. In *Proc. of The 11th Int'l Workshop on Parallel and Distributed Real-time Systems*. IEEE, 2003.

[8] J. Fleischmann, K. Buchenrieder, and R. Kress. Java driven codesign and prototyping of networked embedded systems. In *Proc. of the 36th ACM/IEEE Design Automation Conf.*, pages 794–797, New York, NY, USA, 1999. ACM Press.

[9] S. Friedman, P. Krishnamurthy, R. Chamberlain, R. K. Cytron, and J. E. Fritts. Dusty caches for reference counting garbage collection. In *Proc. of Workshop on Memory Performance: Dealing with Applications, Systems and Architecture*, Sept. 2005.

[10] R. Hough, P. Jones, S. Friedman, R. Chamberlain, J. Fritts, J. Lockwood, and R. Cytron. Cycle-accurate microarchitecture performance evaluation. In *Proc. of Workshop on Introspective Architecture*, Feb. 2006.

[11] J. Irwin, D. May, H. L. Muller, and D. Page. Predictable instruction caching for media processors. In

13th Int'l Conf. on Application-Specific Systems, Architectures and Processors, pages 141–150. IEEE Computer Society Press, July 2002.

[12] P. Jones, S. Padmanabhan, D. Rymarz, J. Maschmeyer, D. V. Schuehler, J. W. Lockwood, and R. K. Cytron. Liquid architecture. In *Proc. of Workshop on Next Generation Software*, Apr. 2004.

[13] P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, and J. Lancaster. Biosequence similarity search on the Mercury system. In *Proc. of the IEEE 15th Int'l Conf. on Application-Specific Systems, Architectures and Processors*, pages 365–375, Sept. 2004.

[14] Free Hardware and Software Resources for System on Chip. `http://www.leox.org`.

[15] J. W. Lockwood. Evolvable Internet hardware platforms. In *The Third NASA/DoD Workshop on Evolvable Hardware*, pages 271–279, July 2001.

[16] T. Nakajima and M. Iwasaki. Issues for making Linux predictable. In *Proc. of Symp. on Applications and the Internet*, Jan. 2002.

[17] S. Padmanabhan, R. K. Cytron, R. D. Chamberlain, and J. W. Lockwood. Application-specific automatic microarchitecture reconfiguration. In *Proc. of 13th Reconfigurable Architectures Workshop*, Apr. 2006.

[18] S. Padmanabhan, P. Jones, D. V. Schuehler, S. J. Friedman, P. Krishnamurthy, H. Zhang, R. Chamberlain, R. K. Cytron, J. Fritts, and J. W. Lockwood. Extracting and improving microarchitecture performance on reconfigurable architectures. In *Proc. of Workshop on Compilers and Tools for Constrained Embedded Systems*, Sept. 2004.

[19] S. Padmanabhan, P. Jones, D. V. Schuehler, S. J. Friedman, P. Krishnamurthy, H. Zhang, R. Chamberlain, R. K. Cytron, J. Fritts, and J. W. Lockwood. Extracting and improving microarchitecture performance on reconfigurable architectures. *Int'l Journal of Parallel Programming*, 33(2–3):115–136, June 2005.

[20] R. Plackett and J. Burman. The design of optimum multifactorial experiments. *Biometrika*, 33(4):305–325, June 1946.

[21] D. V. Schuehler, B. C. Brodie, R. D. Chamberlain, R. K. Cytron, S. J. Friedman, J. Fritts, P. Jones, P. Krishnamurthy, J. W. Lockwood, S. Padmanabhan, and H. Zhang. Microarchitecture optimization for embedded systems. In *Proc. of 8th High Performance Embedded Computing Workshop*, Sept. 2004.

[22] T. Sherwood, E. Perelman, and B. Calder. Block distribution analysis to find periodic behavior and simulation points in applications. In *Proc. of the 2001 Int'l Conf. on Parallel Architectures and Compilation Techniques*, September 2001.

[23] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja, and V. S. Pai. Challenges in computer architecture evaluation. *IEEE Computer*, 36(8):30–36, August 2003.

[24] R. Srinivasan, J. Cook, and S. Cooper. Fast, accurate microarchitecture simulation using statistical phase detection. In *Proc. of the 2005 IEEE Int'l Symposium on Performance Analysis of Systems and Software*, pages 147–156, March 2005.

[25] μClinux – Embedded Linux/Microcontroller Project. `http://www.uClinux.org`.

[26] J. J. Yi, D. J. Lilja, and D. M. Hawkins. Improving computer architecture simulation methodology by adding statistical rigor. *IEEE Transactions on Computers*, 54(11):1360–1373, November 2005.