Empirical Performance Assessment Using Soft-Core Processors on Reconfigurable Hardware*

Richard Hough
Washington University
St. Louis, Missouri
rh3@wustl.edu

Praveen Krishnamurthy
Washington University
St. Louis, Missouri
praveen@wustl.edu

Roger D. Chamberlain
Washington University
St. Louis, Missouri
roger@wustl.edu

Ron K. Cytron
Washington University
St. Louis, Missouri
cytron@wustl.edu

John Lockwood
Washington University
St. Louis, Missouri
lockwood@wustl.edu

Jason Fritts
Saint Louis University
St. Louis, Missouri
jfritts@slu.edu

ABSTRACT

Simulation has been the de facto standard method for performance evaluation of newly proposed ideas in computer architecture for many years. While simulation allows for theoretically arbitrary fidelity (at least to the level of cycle accuracy) as well as the ability to monitor the architecture without perturbing the execution itself, it suffers from low effective fidelity and long execution times.

We (and others) have advocated the use of empirical experimentation on reconfigurable hardware for computer architecture performance assessment. In this paper, we describe an empirical performance assessment subsystem implemented in reconfigurable hardware and illustrate its use. Results are presented that demonstrate the need for the types of performance assessment that reconfigurable hardware can provide.

Categories and Subject Descriptors

B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; C.4 [**Performance of Systems**]: Measurement Techniques

General Terms

Design, Experimentation, Measurement, Performance

1. INTRODUCTION

In computer architecture research, the primary tool used today for performance analysis is simulation. Using simulators such as SimpleScalar [4], SimOS [20], or M5 [5], investigators execute code from common benchmarks (e.g., SPEC [23], MiBench [10], MediaBench [13], CommBench [25]) to assess the performance impact of the architectural features they are interested in evaluating. This reliance on simulation is primarily due to the fact that constructing a physical prototype of a new architecture is cost prohibitive.

Simulation, however, is limited by the following concerns:

- fidelity – simulations are typically performed with abstract, incomplete, or missing components. While it is conceptually easy to describe a "cycle accurate" simulation model, there is little hope of any implementation reflecting the details of that simulation exactly.

- execution time – simulations are typically interpretive so that they can perform sufficient introspection to collect data of interest. As such, they typically run orders of magnitude slower than the system they are modeling.

One viable remediation for the above concerns is to simulate portions of an application's execution by sampling [26]: some segments are simulated in fine detail while the details of executing the other segments are largely ignored. For average behavior, sampling may provide adequate resolution in a reasonable timeframe. However, if an execution contains infrequent events whose details are important, sampling may miss the most noteworthy phenomena.

Moreover, for some applications, occasional worst-case behavior can be more significant than the application's average-case behavior. For example, in the real-time application we consider below, its worst-case execution time is necessary for proper scheduling within the application. A rare event that can lead to increased execution time can adversely affect scheduling or perhaps cause a real-time program to miss a crucial deadline.

Reconfigurable hardware, in the form of field-programmable gate arrays (FPGAs), can be used to model systems that

will ultimately be implemented in custom silicon. In fact, soft-core descriptions of common architecture implementations are becoming widely available. With the appropriate instrumentation of such descriptions, and the addition of logic to log events reliably, execution details at the microarchitectural level can be captured at full (FPGA) speed for an application's entire execution [11].

While much of a program's observed behavior is intrinsic to the application and its host architecture, other processes— some of which may be required for proper system operation— can adversely affect the primary application's performance. System processes responsible for resource management, device allocation, page management, and logging typically execute outside the domain of an application's performance introspection. Thus, debugging and performance monitoring tools can reveal much about about problems within an application, but they are hard pressed to identify interprocess behavior that contributes to poor performance. Moreover, as we show in this paper, interprocess behavior can even mask performance problems within an application.

The use of FPGAs for performance monitoring has recently received a fair amount of attention. In terms of functionality, our approach resembles SnoopP [22], in that we both augment a soft-core architecture (e.g., Microblaze for SnoopP) with logic to capture information based on instruction ranges. Our model, however, utilizes additional logic that allows users to correlate event behavior with the program counter, specific instruction address ranges, and also the process IDs in the operating system. The recently initiated RAMP [3, 19] project uses FPGA technology to perform architectural performance analysis, especially focusing on parallel computing architectures, but they have not yet described the specific mechanisms they intend to use. In a similar vein, IBM's Cell processor has extensive on-chip mechanisms for performance monitoring [9], of course limited to the specific architecture of the Cell processor itself.

In this paper, we make the case that empirical measurement of computer architectures deployed on FPGAs has significant advantages over performance evaluation via simulation, and we describe an infrastructure we have constructed for this purpose, illustrating its use in several experiments.

2. EXPERIMENTAL SYSTEM

We developed an experimental system as part of the *liquid architecture* project [6, 12, 18], utilizing the Field-programmable Port Extender (FPX) platform [17] as our infrastructure basis. The FPX provides a proven environment for interfacing FPGA designs with off-chip memory and a gigabit-rated network interface. Working within this system, we have successfully deployed the LEON2 [8, 14], a SPARC V8 compatible soft-core processor, on the FPX and interfaced it with a memory controller unit, a custom network control processor, and a programmable statistics module (described below). Tailored, application-specific functional acceleration modules are interfaced to the CPU via either the AMBA high-speed bus (AHB) [2], the AMBA peripheral bus (APB), or the standard SPARC co-processor interface. Current OS support includes both uClinux [24] and, with the addition of a memory management unit, the Linux 2.6.11 kernel. Figures 1 and 2 illustrate the liquid architecture system.
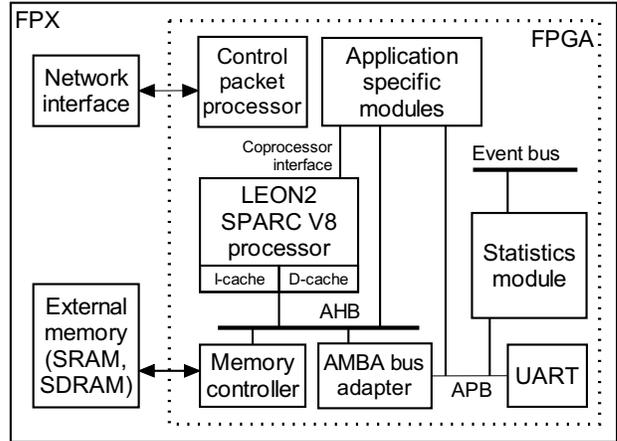

Figure 1: Liquid architecture block diagram.

Figure 2: Liquid architecture photograph.

2.1 Statistics Module Architecture

The statistics module exists as a custom VHDL core resident within the FPGA. Communications to and from the processor are handled by a wrapper interface connected to the APB. In addition to this standard interface, the statistics module receives processor state information via a custom *event bus*. This event bus essentially serves as a point-to-point connection between the statistics-capturing engine and various architectural "hooks" placed throughout the system. Currently our event bus is configured to carry information regarding the state of the instruction cache, data cache, and PC address; however, as the LEON2 is an open processor, one could easily register new hooks to monitor any given architectural feature with minimal modifications to the VHDL.

The statistics-capturing engine is designed to count designated events, as described above, when they occur within a specified region of the executing program. A region is defined as a particular address range for the program counter, and the program's load map (with a suitable GUI) assists a developer in specifying address ranges of interest.

The statistics module provides a distinct advantage for gathering data in that it is fully programmable by the user at run time; the bitfile does not need to be resynthesized to accommodate new combinations of address ranges and events.

As hardware build cycles take approximately 50 minutes on our development machines, this results in a large increase in productivity. To operate the statistics module, the user must program the collection mechanism with the following three different values: the address range of interest, a timer duration between statistic collection, and the 32-bit counter that should be associated with a particular address range and event. By allowing the free association of counters, events, and address ranges, we are able to cover every possible instrumentation combination while utilizing a relatively small portion of the chip resources [11]. Once the user has instrumented their design by communicating the necessary information over the APB bus, the program will be executed and the module alerted. During the program's operation the statistics module will increment its internal counters that track each desired combination of events and address ranges, and store the resulting values for later retrieval every time the user-timer expires.

2.2 Operating System Operation

The instrumentation necessary for our purposes must track performance both within and outside of the application at hand. In this section we describe extensions to the statistics module that accommodate performance profiling among processes running under an operating system. Such an environment requires consideration of two additional factors: the effect of the MMU and the scheduling of multiple competing processes. Fortunately, the former reduces to a simple matter of merely changing the address ranges that the module should watch for a given program. However, since each program shares the same virtual memory address space, a distinction between processes must be instituted to isolate individual program statistics.

To accommodate this, the user can associate a particular counter with a specific Process ID (PID). Counters with an associated PID will only increment if the corresponding process is in control of the CPU. If the user does not assert an association, the given counter will increment regardless of the current PID. A modification was made to the Linux scheduler so that, just before the scheduler finishes switching to a new process, it writes the PID of that process over the APB to the module.

Further work was also necessary to create a Linux driver for communicating to the statistics module when running the OS. In this instance we opted to create a simple character driver, with reads to and writes from the statistics module transferred via calls to ioctl.

2.3 PID Logging

Due to the competitive nature of thread scheduling in the Linux kernel, the variance in statistical results from one run to the next is greater than one would find when running a stand alone application. In particular, one could potentially see large differences in execution time if an infrequent process, such as a kernel daemon, should be scheduled in one experimental run but not the other.

To assist in tracking down these rare events, a PID log was added to the module. When the statistics module is started, a 32-bit PID timer is continually incremented. Once the scheduler writes the PID of the new running thread to the statistics module, the value of the PID counter and the new PID are stored into a BlockRAM FIFO. The PID timer is then cleared, and the counting continues until the next context switch by the scheduler. At the end of the run the user may then read back the log and get a clock-cycle accurate picture of the order and duration of the context switches within their system.

3. DESCRIPTION OF EXPERIMENTS

An important consideration when attempting any form of performance analysis is whether or not the experiment correctly reflects the essential (performance impacting) characteristics of the target system that is being modeled. With simulation-based performance analysis, there is an inherent tradeoff between the complexity of the system that can be modeled and the fidelity of the underlying model itself.

As a result, existing simulation models must make choices as to where they position themselves in this space. For example, very high fidelity models that approach (or achieve) cycle-accurate fidelity are constrained as to the scope of the model, often modeling single threads of execution running standalone on a processor without an operating system and certainly without additional competing applications present. At the other end of the spectrum, models that include the full operating system will simplify (to some degree) the underlying model fidelity. Of course, there are numerous examples between these two extremes as well.

The use of reconfigurable hardware to address this performance analysis limitation has been proposed by a number of groups. In this section, we illustrate the capabilities of our performance analysis system and describe its benefits. First, we show how the complexity of the modeled system can be increased (by including both the operating system as well as competing applications) without diminishing the fidelity of the model. Second, we demonstrate the ability to reason about and quantitatively investigate rare events that are particularly difficult to address with simulation modeling because of the long execution times involved.

3.1 Process-Centric Measurements

When proposing some new (or altered) microarchitectural feature, it is standard practice to empirically evaluate the efficacy of that feature across a set of benchmark applications. Similarly, the microarchitectural feature might not be new at all, in an embedded system design we might simply be interested in choosing the parameter setting that best suits the needs of the current application. Here, we illustrate the ability to do this type of investigation by measuring cache hit rates as a function of cache size and associativity for a set of benchmark applications (most of which are traditionally embedded applications). Our particular interest here is the degree to which the performance measures are altered by the presence of both an operating system and other applications present and competing for processor resources.

First we describe the benchmark set, followed by the experimental procedure.

3.1.1 Benchmarks

The MiBench benchmark suite [10] consists of a set of embedded system workloads which differ from standard desk-

top workloads. The applications contained in the MiBench suite were selected to capture the diversity of workloads in embedded systems. For the purposes of this study, we chose workloads from the networking, telecommunications, and automotive sections of the suite.

CommBench [25] was designed with the goal of evaluating and designing telecommunications network processors. The benchmark consists of 8 programs, 4 of which focus on packet header processing, and the other 4 are geared towards data stream processing.

Following are the set of applications we have used as part of this study:

- From MiBench:

  - *basicmath*: This application is part of the automotive applications inside MiBench. It computes cubic functions, integer square roots and angle conversions.
  - *sha*: This is part of the networking applications inside MiBench. Sha is a secure hash algorithm which computes a 160-bit digest of inputs.
  - *fft*: This is part of the telecommunication applications inside MiBench, and computes the fast Fourier transform on an array of data.

- From CommBench:

  - *drr*, *frag*: These are part of the header processing apps inside CommBench. The drr algorithm is used for bandwidth scheduling for large numbers of flows. Frag refers to the fragmentation algorithm used in networking to split IP packets.
  - *reed_enc*, *reed_dec*: These are part of the packet processing applications in CommBench. They are the encoder and decoder used in the Reed-Solomon forward error correction scheme.

To the above set we added one locally developed benchmark, *blastn*, which implements the first (hashing) stage of the popular BLAST biosequence alignment application for nucleotide sequences [1].

3.1.2 Procedure
For each of the applications, the following sequence of actions was taken:

- The application was executed under the Linux 2.6.11 OS on the Liquid Architecture system. The OS was in single-user mode with no other user applications enabled. The benchmark was adapted to initiate the statistics collection subsystem (this required the addition of a single call at the beginning of the code).

- A subset of the applications was executed with one or more competing applications concurrently scheduled. The competing applications were drawn from the same benchmark set. Figure 3 shows the pairing of primary and competing applications.

primary application	single competing application	set of 3 competing applications
drr	*frag*	—
frag	*reed_dec*	—
reed_dec	*reed_dec*	—
sha	*fft*	—
blastn	*fft*	*fft, drr, frag*
fft	*reed_enc*	*reed_enc, reed_dec, fft*

Figure 3: Pairings of primary and competing applications.

- A number of different configurations of the LEON processor were generated. Data cache sizes of 2, 4, 8, and 16 Kbytes were included for a two-way associative cache, and cache sizes of 4, 8, and 16 Kbytes were included for a four-way associative cache.

- Sets of the applications were executed on each of the processor configurations, measuring loads, stores, cache hits, cache misses, memory reads, and memory writes. It is the variations in these parameters that we wish to examine.

Each execution was repeated five times. The mean results are reported in the tables, and the plots include both mean and error bars representing 95% confidence intervals.

3.2 Rare Event Investigation
While the above section emphasizes the exploration of an architectural design space, we next concentrate on the need to investigate rare events. This is first motivated by a specific case study, which is followed by an illustration of the use of the statistics module to perform this type of investigation.

3.2.1 Motivation
In this section we present a case study that motivates the techniques we present in this paper. Real-time applications often involve tasks that must be scheduled so as to know they will complete within a given timeframe. The analysis [16] required to prove that deadlines are met necessitates knowing the cost (time) of the code that must be scheduled within the tasks, as well as the tasks' deadlines and periodicity. Static scheduling analysis [15] requires a worst-case bound on the tasks' costs, so that scheduling can account for worst-case behavior to ensure that deadlines are met.

Consider a given task's worst-case and average-case execution time. For real-time scheduling, the time alloted for each release of the task must account for the task's worst-case behavior. Its actual utilization will follow the task's average-case cost, but statically it cannot be determined when the task will experience average- or worst-case behavior. Thus, the task most suitable for real-time applications has an average cost that is nearly the task's worst-case cost.

Consider a simple hash table, into which data will be inserted and retrieved by a real-time application. The cost of a "put" into the table is typically quite small. However,

most implementations test the capacity of a hash table during a `put` operation; if the table should be resized, then the table's reorganization is accomplished during the `put`. Thus, the cost of some `put` operations can be much worse than the `put`'s average cost.

Real-time implementations of hash tables [7] amortize the excessive cost over all `put` operations, so that the hash table adapts slightly at every `put` and the cost of each `put` is *theoretically* the same. Execution of such an implementation is shown in Figure 4 for ∼5,000 `put` operations. The data was collected under Solaris 8 on a Sparc 5 with the application running at the highest priority in real-time mode; no other task supposedly could pre-empt the application. Note that almost every `put` operation is within 980 nanoseconds. Occasionally, a `put` is observed to be significantly more expensive and can take as much as ∼23 microseconds.
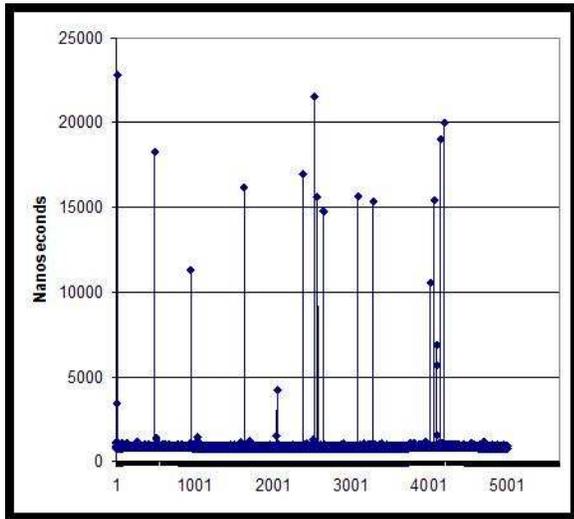
Figure 4: Observed execution times for a real-time HashTable `put` operation

Following are results obtained via classical approaches for determining the source of the excessive execution times:

- The code can be instrumented within the `put` to determine which statement or segment of code is responsible for the observed times.

  Standard tools do not instrument at such a level, but manual insertion of timers revealed that the problem could not be attributed to any one section of the code.

- Cache and other such features can be disabled or made useless to determine if the problem arises at a microarchitectural level.

  With the cache effectively disabled, the execution times were uniformly worse (as expected) but there were still occasional `put` operations whose times greatly exceeded the average case.

Based on the unpredictability of the worst-case observed execution times, it is clear that the spikes in Figure 4 are due to activity occurring in other processes or threads that cause

the CPU to be taken from the real-time task. In theory, such activity should not occur: the application executed in "real-time" mode on an operating system (Solaris) that supposedly supports such applications, and all pages were locked into memory.

Because the events in Figure 4 occur rarely and seemingly at random, sampling methods are likely to miss the moment of bad behavior. Moreover, the problem exists between two separate address spaces and between processes that may not have permission to inspect each other. Finally, the code segment of interest is relatively brief; any method for finding the source of the bad behavior must be sufficiently nonintrusive so as not contribute to or mask the actual bad behavior.
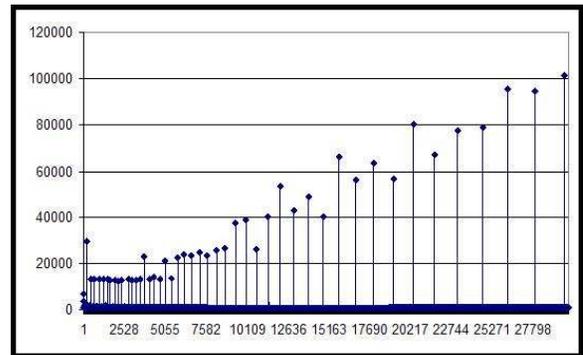
Figure 5: Isolated execution times for HashTable `put`

If the sources of the unpredictable behavior are located, using the methods we propose in this paper, then the application's behavior per `put` is shown in Figure 5. Note that the data includes data points beyond the first 5000 shown in Figure 4. While the times still do not reflect the desired real-time behavior, the pattern of spikes is now much clearer and was easily resolved to a problem with the storage allocator. When more storage is necessary for expansion of the hash table, the allocator is spending nonlinear time, which contributes to poor real-time performance. By substituting the ACE allocator [21], we obtain the performance shown in Figure 6.
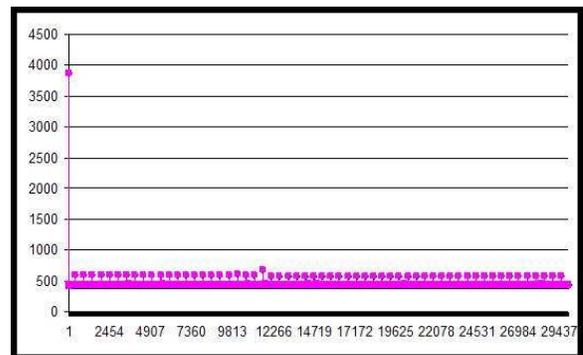
Figure 6: Real-time performance obtained with a better allocator

In summary, this case study illustrates the difficulties faced in obtaining an accurate picture of an application's performance when that performance is adversely affected by other processes. Standard debugging and profiling tools are unable to capture system-wide performance data at a resolution and on a scale that allows a developer to appreciate the application's behavior. Indeed, in this case, system interference with the application masked an actual problem in the application that was easily fixed (if not easily found) to obtain real-time performance.

3.2.2   Rare Event Experiment

To illustrate the abilities of the statistics module for investigating events of the type just described, we repeatedly executed the *blastn* application 549 times and measured the total execution times shown in Figure 7. Note that the vast majority of the runs complete in 2.717 billion clock cycles, several runs take an additional 3 million clock cycles, but 15 runs take an additional 5.5 million clock cycles to finish. We configured the statistics module to investigate the properties of these incremental 5.5 million clock cycles.
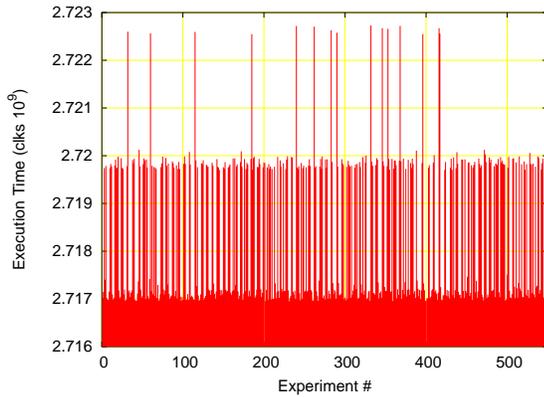
Figure 7: Total execution time for *blastn* application, multiple runs.

4.   RESULTS

The results are described in two sections, starting with the discussion of the cache behavior and following with the rare event investigation.

4.1   Cache Behavior

Figures 8 to 10 show the mean execution time, data cache read miss rates, and data cache write miss rates for the benchmark applications when executing code in the virtual address range 0 to 0x7FFFFFFF (i.e., the .text segment of the application itself, excluding all system calls). Each of these values is presented twice. The first is for the entire execution (i.e., application, OS, and any competing applications) and the second is for the application alone (i.e., configuring the statistics module to be sensitive only to the primary application process).

The ability to collect this data illustrates the discriminatory features of the statistics module, both restricting the PID to that of the application and also restricting the address space of the data collection within the application. In addition, the ability to cover this wide of a design space is significantly

enabled by the fact that the investigation is executing on an FPGA rather than a simulation model. The execution time required to collect all of the data comprised approximately 4 trillion clock cycles, requiring approximately 40 hours of FPGA execution time. This would have been simply prohibitive in a software simulation environment.
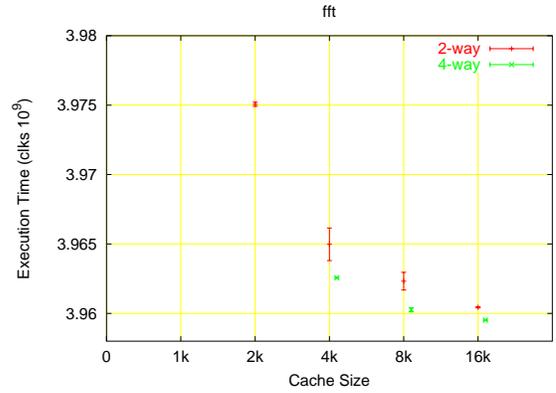
Figure 11: Execution time (in billions of clock cycles) for *fft* running on OS with no other competing application. Various dcache configurations are shown.
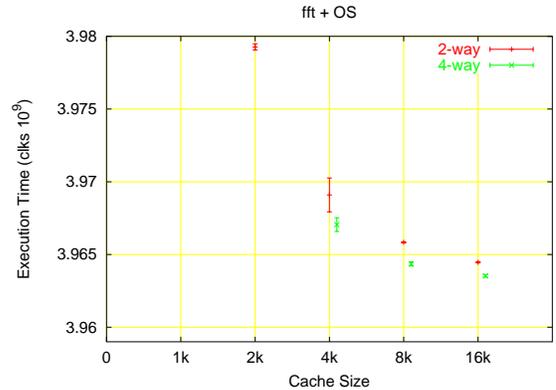
Figure 12: Execution time (in billions of clock cycles) for total of *fft* plus the OS with no other competing application. Various dcache configurations are shown.

The next set of graphs illustrate some of the interesting features of this data set. Figure 11 shows the application-only execution time for *fft* for varying dcache configurations when running on the OS without a competing application. Figure 12 shows the complete execution time (*fft* application plus OS) for the same set of experiments. Note that there is only a slight increase in execution time across the board, implying that the OS is not significantly impacting the execution of the application (i.e., at each scheduling quantum, the scheduler simply returns to the application). Figure 13 plots *fft*-only execution time when there is a competing application (*reed_enc*) scheduled concurrently. Note the similarity to Figure 11, indicating that the competing application doesn't significantly impact the execution time required for *fft* alone. Contrast this with Figure 14, which plots the total execution time for all of *fft*, the competing application

Application	size, assoc.	os+app (10^9 clks)	app (10^9 clks)	os+2apps (10^9 clks)	app (10^9 clks)	os+4apps (10^9 clks)	app (10^9 clks)
basicmath	2K, 2-way	8.468	8.458	—	—	—	—
	4K, 2-way	8.453	8.444	—	—	—	—
	8K, 2-way	8.437	8.428	—	—	—	—
	16K, 2-way	8.432	8.424	—	—	—	—
	4K, 4-way	8.443	8.433	—	—	—	—
	8K, 4-way	8.435	8.426	—	—	—	—
	16K, 4-way	8.431	8.423	—	—	—	—
reed_enc	2K, 2-way	2.024	2.022	—	—	—	—
	4K, 2-way	1.935	1.933	—	—	—	—
	8K, 2-way	1.924	1.922	—	—	—	—
	16K, 2-way	1.921	1.919	—	—	—	—
	4K, 4-way	1.933	1.931	—	—	—	—
	8K, 4-way	1.923	1.920	—	—	—	—
	16K, 4-way	1.921	1.919	—	—	—	—
drr	2K, 2-way	2.706	2.703	5.243	2.704	—	—
	4K, 2-way	2.650	2.647	5.180	2.647	—	—
	8K, 2-way	2.618	2.615	5.147	2.614	—	—
	16K, 2-way	2.605	2.602	5.134	2.603	—	—
	4K, 4-way	2.646	2.644	5.176	2.644	—	—
	8K, 4-way	2.611	2.608	5.140	2.608	—	—
	16K, 4-way	2.605	2.603	5.134	2.603	—	—
frag	2K, 2-way	2.541	2.538	5.081	2.537	—	—
	4K, 2-way	2.535	2.531	5.081	2.532	—	—
	8K, 2-way	2.533	2.530	5.084	2.530	—	—
	16K, 2-way	2.532	2.530	5.077	2.529	—	—
	4K, 4-way	2.533	2.531	5.091	2.530	—	—
	8K, 4-way	2.533	2.530	5.075	2.530	—	—
	16K, 4-way	2.532	2.530	5.077	2.530	—	—
reed_dec	2K, 2-way	4.682	4.677	9.349	4.675	—	—
	4K, 2-way	4.614	4.599	9.214	4.604	—	—
	8K, 2-way	4.603	4.591	9.191	4.595	—	—
	16K, 2-way	4.581	4.576	9.153	4.577	—	—
	4K, 4-way	4.603	4.598	9.200	4.600	—	—
	8K, 4-way	4.586	4.581	9.148	4.580	—	—
	16K, 4-way	4.582	4.578	9.135	4.578	—	—
sha	2K, 2-way	9.166	9.157	13.129	9.145	—	—
	4K, 2-way	9.141	9.119	13.100	9.122	—	—
	8K, 2-way	9.123	9.111	13.067	9.097	—	—
	16K, 2-way	9.115	9.106	13.044	9.080	—	—
	4K, 4-way	9.127	9.117	13.078	9.109	—	—
	8K, 4-way	9.119	9.107	13.054	9.090	—	—
	16K, 4-way	9.113	9.103	13.041	9.074	—	—
blastn	2K, 2-way	4.497	4.492	8.474	4.493	13.715	4.491
	4K, 2-way	4.488	4.484	8.453	4.484	13.633	4.482
	8K, 2-way	4.482	4.477	8.443	4.478	13.590	4.477
	16K, 2-way	4.468	4.463	8.427	4.464	13.562	4.464
	4K, 4-way	4.480	4.476	8.442	4.476	13.618	4.475
	8K, 4-way	4.474	4.469	8.433	4.469	13.573	4.469
	16K, 4-way	4.470	4.465	8.428	4.465	13.562	4.465
fft	2K, 2-way	3.979	3.975	6.001	3.976	13.948	3.981
	4K, 2-way	3.969	3.965	5.898	3.966	13.839	3.967
	8K, 2-way	3.966	3.962	5.885	3.963	13.824	3.964
	16K, 2-way	3.964	3.960	5.881	3.961	13.828	3.962
	4K, 4-way	3.967	3.963	5.895	3.963	13.846	3.964
	8K, 4-way	3.964	3.960	5.882	3.961	13.828	3.960
	16K, 4-way	3.964	3.960	5.880	3.960	13.824	3.960

Figure 8: Execution time results for 8 benchmark applications.

Application	size, assoc.	os+app miss rate	app miss rate	os+2apps miss rate	app miss rate	os+4apps miss rate	app miss rate
basicmath	2K, 2-way	0.025	0.024	—	—	—	—
	4K, 2-way	0.018	0.018	—	—	—	—
	8K, 2-way	0.011	0.011	—	—	—	—
	16K, 2-way	0.010	0.009	—	—	—	—
	4K, 4-way	0.014	0.013	—	—	—	—
	8K, 4-way	0.011	0.011	—	—	—	—
	16K, 4-way	0.009	0.009	—	—	—	—
reed_enc	2K, 2-way	0.043	0.043	—	—	—	—
	4K, 2-way	0.011	0.011	—	—	—	—
	8K, 2-way	0.007	0.007	—	—	—	—
	16K, 2-way	0.006	0.006	—	—	—	—
	4K, 4-way	0.010	0.010	—	—	—	—
	8K, 4-way	0.006	0.006	—	—	—	—
	16K, 4-way	0.006	0.006	—	—	—	—
drr	2K, 2-way	0.082	0.082	0.051	0.082	—	—
	4K, 2-way	0.044	0.044	0.029	0.044	—	—
	8K, 2-way	0.023	0.022	0.017	0.022	—	—
	16K, 2-way	0.014	0.014	0.012	0.015	—	—
	4K, 4-way	0.042	0.042	0.027	0.042	—	—
	8K, 4-way	0.018	0.018	0.014	0.018	—	—
	16K, 4-way	0.014	0.015	0.012	0.014	—	—
frag	2K, 2-way	0.016	0.016	0.024	0.015	—	—
	4K, 2-way	0.012	0.011	0.013	0.011	—	—
	8K, 2-way	0.010	0.010	0.011	0.010	—	—
	16K, 2-way	0.009	0.010	0.009	0.009	—	—
	4K, 4-way	0.010	0.010	0.011	0.010	—	—
	8K, 4-way	0.010	0.010	0.009	0.010	—	—
	16K, 4-way	0.009	0.009	0.009	0.009	—	—
reed_dec	2K, 2-way	0.029	0.029	0.029	0.029	—	—
	4K, 2-way	0.015	0.012	0.014	0.013	—	—
	8K, 2-way	0.012	0.011	0.011	0.011	—	—
	16K, 2-way	0.008	0.007	0.008	0.008	—	—
	4K, 4-way	0.012	0.012	0.012	0.013	—	—
	8K, 4-way	0.009	0.009	0.008	0.008	—	—
	16K, 4-way	0.008	0.008	0.008	0.008	—	—
sha	2K, 2-way	0.007	0.007	0.008	0.006	—	—
	4K, 2-way	0.006	0.005	0.006	0.005	—	—
	8K, 2-way	0.005	0.005	0.005	0.004	—	—
	16K, 2-way	0.004	0.004	0.003	0.003	—	—
	4K, 4-way	0.005	0.005	0.005	0.005	—	—
	8K, 4-way	0.005	0.004	0.004	0.004	—	—
	16K, 4-way	0.004	0.004	0.003	0.003	—	—
blastn	2K, 2-way	0.035	0.035	0.033	0.035	0.039	0.035
	4K, 2-way	0.033	0.033	0.029	0.033	0.029	0.033
	8K, 2-way	0.031	0.031	0.027	0.031	0.023	0.031
	16K, 2-way	0.028	0.028	0.024	0.028	0.020	0.028
	4K, 4-way	0.031	0.031	0.027	0.031	0.027	0.031
	8K, 4-way	0.029	0.029	0.025	0.029	0.021	0.029
	16K, 4-way	0.028	0.028	0.024	0.028	0.020	0.028
fft	2K, 2-way	0.023	0.023	0.037	0.024	0.030	0.028
	4K, 2-way	0.015	0.015	0.011	0.016	0.013	0.017
	8K, 2-way	0.012	0.013	0.008	0.013	0.010	0.014
	16K, 2-way	0.011	0.011	0.007	0.011	0.008	0.012
	4K, 4-way	0.013	0.013	0.011	0.013	0.012	0.014
	8K, 4-way	0.011	0.011	0.007	0.011	0.009	0.011
	16K, 4-way	0.010	0.010	0.007	0.010	0.008	0.011

Figure 9: Dcache read miss rate results for 8 benchmark applications.

Application	size, assoc.	os+app miss rate	app miss rate	os+2apps miss rate	app miss rate	os+4apps miss rate	app miss rate
basicmath	2K, 2-way	0.008	0.007	—	—	—	—
	4K, 2-way	0.005	0.005	—	—	—	—
	8K, 2-way	0.002	0.002	—	—	—	—
	16K, 2-way	0.002	0.002	—	—	—	—
	4K, 4-way	0.004	0.003	—	—	—	—
	8K, 4-way	0.003	0.002	—	—	—	—
	16K, 4-way	0.002	0.002	—	—	—	—
reed_enc	2K, 2-way	0.015	0.015	—	—	—	—
	4K, 2-way	0.007	0.007	—	—	—	—
	8K, 2-way	0.006	0.006	—	—	—	—
	16K, 2-way	0.006	0.005	—	—	—	—
	4K, 4-way	0.007	0.006	—	—	—	—
	8K, 4-way	0.006	0.006	—	—	—	—
	16K, 4-way	0.006	0.005	—	—	—	—
drr	2K, 2-way	0.038	0.038	0.041	0.039	—	—
	4K, 2-way	0.035	0.034	0.036	0.034	—	—
	8K, 2-way	0.032	0.031	0.033	0.031	—	—
	16K, 2-way	0.028	0.027	0.031	0.028	—	—
	4K, 4-way	0.034	0.034	0.035	0.034	—	—
	8K, 4-way	0.032	0.031	0.033	0.031	—	—
	16K, 4-way	0.028	0.027	0.031	0.028	—	—
frag	2K, 2-way	0.043	0.043	0.024	0.041	—	—
	4K, 2-way	0.038	0.036	0.021	0.036	—	—
	8K, 2-way	0.035	0.034	0.019	0.034	—	—
	16K, 2-way	0.033	0.033	0.018	0.033	—	—
	4K, 4-way	0.035	0.035	0.019	0.034	—	—
	8K, 4-way	0.034	0.034	0.019	0.034	—	—
	16K, 4-way	0.034	0.033	0.018	0.033	—	—
reed_dec	2K, 2-way	0.010	0.010	0.010	0.010	—	—
	4K, 2-way	0.007	0.006	0.007	0.006	—	—
	8K, 2-way	0.006	0.006	0.006	0.006	—	—
	16K, 2-way	0.005	0.005	0.005	0.005	—	—
	4K, 4-way	0.006	0.006	0.006	0.006	—	—
	8K, 4-way	0.006	0.005	0.006	0.005	—	—
	16K, 4-way	0.006	0.005	0.006	0.005	—	—
sha	2K, 2-way	0.006	0.005	0.005	0.004	—	—
	4K, 2-way	0.004	0.002	0.003	0.003	—	—
	8K, 2-way	0.002	0.001	0.002	0.001	—	—
	16K, 2-way	0.001	0.001	0.001	0.001	—	—
	4K, 4-way	0.003	0.002	0.002	0.002	—	—
	8K, 4-way	0.002	0.001	0.001	0.001	—	—
	16K, 4-way	0.001	0.001	0.001	0.001	—	—
blastn	2K, 2-way	0.004	0.004	0.006	0.005	0.017	0.004
	4K, 2-way	0.003	0.002	0.004	0.003	0.014	0.002
	8K, 2-way	0.002	0.002	0.003	0.002	0.012	0.002
	16K, 2-way	0.001	0.001	0.002	0.001	0.011	0.001
	4K, 4-way	0.003	0.002	0.003	0.003	0.013	0.002
	8K, 4-way	0.002	0.002	0.002	0.002	0.012	0.002
	16K, 4-way	0.001	0.001	0.002	0.001	0.011	0.001
fft	2K, 2-way	0.007	0.006	0.012	0.008	0.011	0.011
	4K, 2-way	0.005	0.004	0.006	0.005	0.006	0.005
	8K, 2-way	0.003	0.003	0.005	0.004	0.005	0.003
	16K, 2-way	0.003	0.003	0.004	0.003	0.004	0.003
	4K, 4-way	0.004	0.003	0.005	0.003	0.005	0.004
	8K, 4-way	0.003	0.002	0.004	0.003	0.004	0.003
	16K, 4-way	0.003	0.002	0.004	0.002	0.004	0.002

Figure 10: Dcache write miss rate results for 8 benchmark applications.

(*reed_enc*), and the OS. Here there is clearly an increase in execution time, as expected, due to the significant additional computational requirements associated with both applications vs. just one solo application.
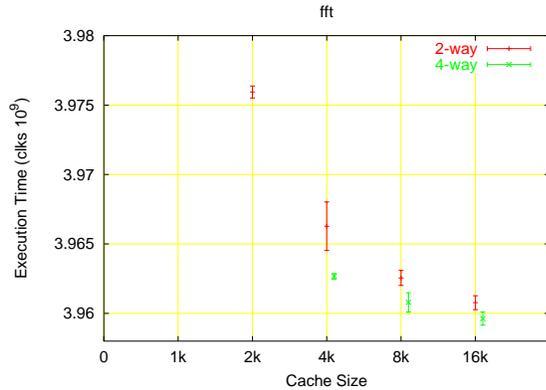

Figure 13: Execution time (in billions of clock cycles) for *fft* running on OS with *reed_enc* as a competing application. Various dcache configurations are shown.
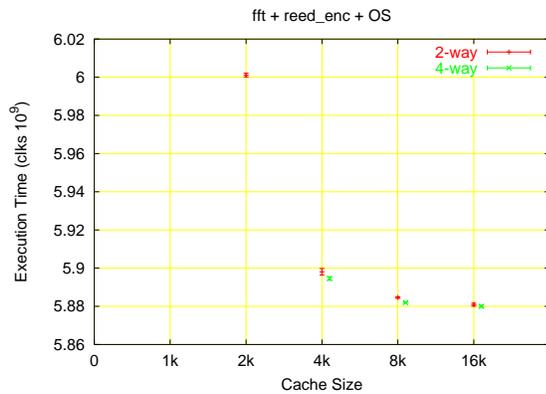

Figure 14: Execution time (in billions of clock cycles) for total of *fft* plus *reed_enc* plus the OS. Various dcache configurations are shown.

With *drr*, the presence of an additional competing application increases the write miss rate for the dcache for a 2 KB and a 16 KB cache size, but does not significantly impact the dcache write miss rate for 4 KB and 8 KB cache sizes. This is shown in Figures 15 and 16.

With *frag*, the presence of the competing application doesn't have a significant impact on the mean dcache read miss rates, but dramatically increases the variability across individual runs, especially near the knee of the curve for the 2-way associative cache. This is shown in Figures 17 and 18.

4.2   Rare Events

For the rare event experiments we used both the event monitoring and the PID logging features of the statistics module. For one cache configuration (4 KB, direct-mapped data cache) we evaluated the variation in the execution time of the *blastn* application.
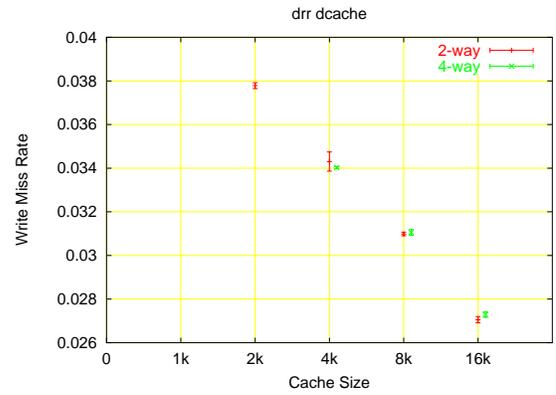

Figure 15: Dcache write miss rate for *drr* running on OS with no other competing application.
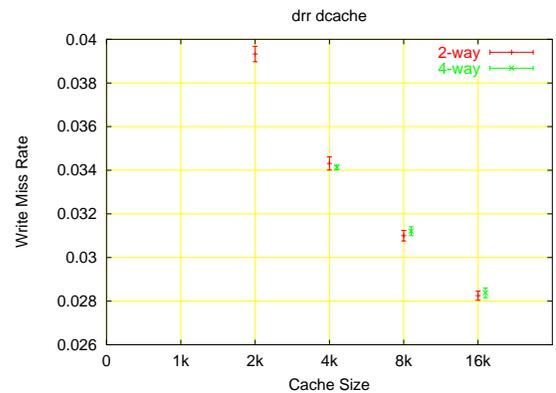

Figure 16: Dcache write miss rate for *drr* with one competing application (*frag*).

Similar to the experiments described in the earlier section, we booted the OS and launched our application to run successively 549 times. Here, *blastn* is the only user application running on the system, and we nominally expect a very low variance in its execution time. For each of the runs we monitored the time spent by the processor in 8 uniform virtual address ranges from 0 to 0xFFFFFFFF. Also, as described in Section 2.3, we kept track of the PID changes within the application run and the time spent between changes. At the end of each run, we can examine the division of execution time between all PIDs run in the system during that window of time represented by the log.

Of the address ranges monitored, we observed execution time attributed to only 3 of the 8 ranges. Figure 7 shows the total execution time of the application over the 549 runs. To investigate the 15 "rare events," application executions taking an additional 5.5 million clock cycles, we start by examining the activity in the 3 active address ranges. These are plotted in Figures 19, 20, and 21. We continue the investigation by examining the PID log for several individual runs. Figure 22 shows this information. Runs 32 and 240 are two of the long runs, and run 50 represents a typical run.

Examination of this data leads us to an important conclusion, the causes of the rare events are not all the same. In
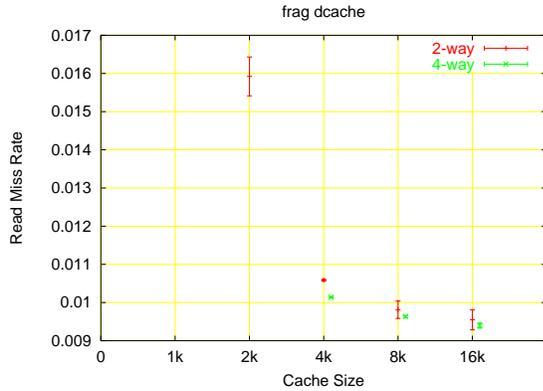
Figure 17: Dcache read miss rate for _frag_ running on OS with no other competing application.
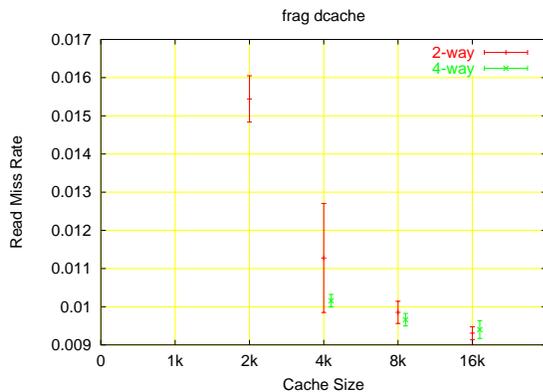

Figure 18: Dcache read miss rate for _frag_ with one competing application (_reed_dec_).

Figure 19: Execution time spent in address range 0 to 0x1FFFFFFF for multiple runs of _blastn_.
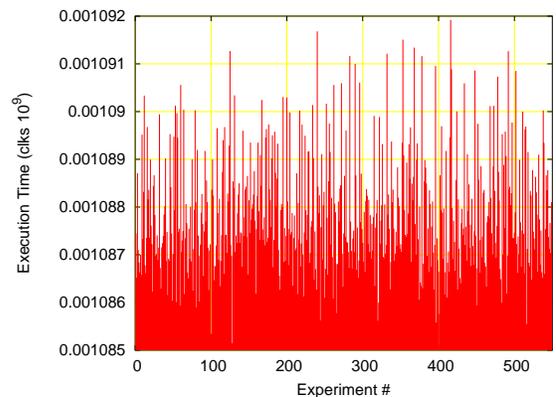
Figure 20: Execution time spent in address range 0x40000000 to 0x5FFFFFFF for multiple runs of _blastn_.

run 32, approximately 2 million additional clock cycles can be attributed to the application itself (a fact that is also true of run 60), and the remaining excess clock cycles are in the kernel (PID 0). For run 240, virtually all of the additional clock cycles are in the kernel, not the application. Furthermore, the distribution of excess clock cycles on the two long runs differs in address range as well. About 2 million additional clocks are present in the low address range for run 32, with the remaining 3.5 million clock cycles in the highest address range (which includes the idle loop). For run 240, all of the additional clock cycles are in the high addresses.

Given the above information, it is next reasonable to parameterize the statistics module for a more focused address range investigation, ultimately identifying the specific methods (whether in the kernel or the application) that account for the excess execution time.

5. CONCLUSIONS

In this paper we have described an approach to empirical performance assessment that exploits the capabilities of FP-GAs. By including monitoring logic on-chip with the processor, it is possible to non-intrusively assess the performance of individual applications in a time-efficient manner.
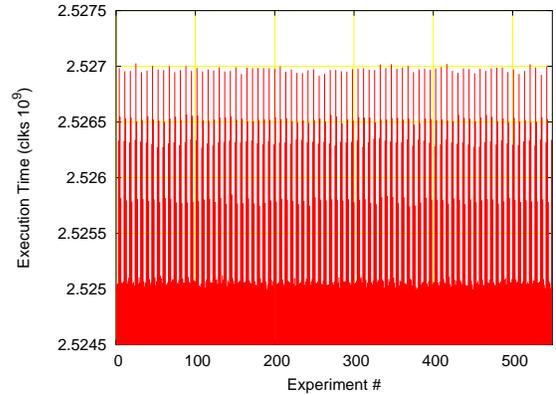
We include two performance assessment examples. The first is a wide characterization of the cache parameterization options, and an investigation of how those results vary in the presence of multiple applications. The second is a rare event investigation, in which mean-value statistics and/or sampling techniques are typically ineffective. The results clearly indicate the benefits of empirical measurement of an FPGA implementation relative to simulation-based performance prediction.

6. REFERENCES

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, et al. Basic local alignment search tool. _Journal of Molecular Biology_, 215:403–10, 1990.

[2] AMBA Specification (Rev 2.0). ARM, Ltd., http://www.arm.com/products/solutions/AMBA_spec.html.

[3] Arvind, K. Asanovic, D. Chiou, J. C. Hoe, C. Kozyrakis, S.-L. Lu, M. Oskin, D. Patterson, J. Rabaey, and J. Wawrzynek. RAMP: Research Accelerator for Multiple Processors – A Community Vision for a Shared Experimental Parallel HW/SW Platform. Technical Report UCB//CSD-05-1412, Sept. 2005.

Experiment number	Process ID					
	0	2	3	8	23	app
32	70,901,766	17538	3,451,566	267,588	4,099,036	2,643,665,649
50	67,517,785	18537	3,337,118	267,970	4,096,603	2,641,632,437
66	67,884,519	16475	3,316,209	268,377	4,160,005	2,643,921,380
240	73,083,068	22986	3,344,711	272,197	4,100,255	2,641,638,131

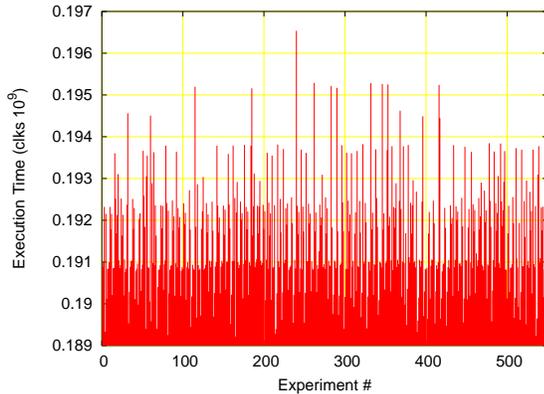**Figure 22: Execution time split across PIDS.**

Figure 21: Execution time spent in address range 0xE0000000 to 0xFFFFFFFF for multiple runs of *blastn*.

[4] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, Feb. 2002.

[5] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-oriented full-system simulation using M5. In *Proc. of 6th Workshop on Computer Architecture Evaluation using Commercial Workloads*, Feb. 2003.

[6] R. D. Chamberlain, R. K. Cytron, J. E. Fritts, and J. W. Lockwood. Vision for liquid architecture. In *Proc. of Workshop on Next Generation Software*, Apr. 2006.

[7] S. Friedman, N. Leidenfrost, B. C. Brodie, and R. K. Cytron. Hashtables for embedded and real-time systems. In *Proceedings of the IEEE Workshop on Real-time Embedded Systems*, 2001.

[8] J. Gaisler. The LEON Processor. www.gaisler.com, 2005.

[9] M. Genden, R. Raghavan, M. Riley, J. Spannaus, and T. Chen. Real-time performance monitoring and debug features of the first generation Cell processor. In *Proc. of 1st Workshop on Tools and Compilers for Hardware Acceleration*, Sept. 2006.

[10] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of IEEE 4th Workshop on Workload Characterization*, 2001.

[11] 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.

[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] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

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

[15] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-time Environment. *JACM*, 20(1):46–61, Jan. 1973.

[16] J. W. S. Liu. *Real-time Systems.* Prentice Hall, New Jersey, 2000.

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

[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. *Int'l Journal of Parallel Programming*, 33(2–3):115–136, June 2005.

[19] D. Patterson, Arvind, K. Asanovic, D. Chiou, J. C. Hoe, C. Kozyrakis, S.-L. Lu, M. Oskin, J. Rabaey, and J. Wawrzynek. RAMP: Research Accelerator for Multiple Processors. In *Hot Chips*, August 2006.

[20] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, Jan. 1997.

[21] D. C. Schmidt. ACE: an Object-Oriented Framework for Developing Distributed Applications. In *Proceedings of the 6th USENIX C++ Technical Conference*, Cambridge, Massachusetts, Apr. 1994. USENIX Association.

[22] L. Shannon and P. Chow. Using reconfigurability to achieve real-time profiling for hardware/software codesign. In *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable*

Gate Arrays, pages 190–199, New York, NY, USA, 2004. ACM Press.

[23] Standard Performance Evaluation Corporation. `http://www.spec.org`.

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

[25] T. Wolf and M. A. Franklin. CommBench – A telecommunications benchmark for network processors. In *Proc. of IEEE Int'l Symp. on Performance Analysis of Systems and Software*, pages 154–162, Austin, TX, Apr. 2000.

[26] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Statistical sampling of microarchitecture simulation. *ACM Trans. Model. Comput. Simul.*, 16(3):197–224, 2006.