Workshop on Compilers and Tools for Constrained Embedded Systems (CTCES);
Washington DC; Sept. 22, 2004

Extracting and Improving Microarchitecture Performance on Reconfigurable Architectures *

Shobana Padmanabhan, Phillip Jones, David V. Schuehler,
Scott J. Friedman, Praveen Krishnamurthy, Huakai Zhang,
Roger Chamberlain, Ron K. Cytron, Jason Fritts, and John W. Lockwood

Department of Computer Science and Engineering
Washington University, St. Louis
Contact email: cytron@acm.org

September 1, 2004

Abstract

We describe our experience using reconfigurable architectures to develop an understanding of an application's performance and to enhance its performance with respect to customized, constrained logic. We begin with a standard ISA currently in use for embedded systems. We modify its core to measure performance characteristics, obtaining a system that provides cycle-accurate timings and presents results in the style of `gprof`, but with absolutely no software overhead. We then provide cache-behavior statistics that are typically unavailable in a generic processor. In contrast with simulation, our approach executes the program at full speed and delivers statistics based on the actual behavior of the cache subsystem. Finally, in response to the performance profile developed on our platform, we evaluate various uses of the FPGA-realized instruction and data caches in terms of the application's performance.

1 Introduction

Embedded applications are held to a higher standard than desktop applications along a number of dimensions. Not only must the functionality of the application be correct, it often must meet strict time constraints and function with restrictive resource limitations (e.g., power, memory). Conversely, the hardware systems used to execute embedded applications are often dedicated to that particular application, alleviating the need to be as general purpose as desktop systems. As a result, there has been significant interest in the ability to build custom hardware platforms for which the design of the hardware is closely matched to the needs of the application.

There are a number of processors available now that can be at least partially configured at the architectural level, such as Tensilica [37] and ARC [2]. While the systems from ARC are configurable at fabrication time, Stretch [35] makes the Tensilica processor reconfigurable at execution time through the use of **Field-Programmable Gate Array** (FPGA) technology on the chip. Academic research into these types of systems is also an active area of investigation [12, 19, 26, 21].

In this paper we present tools and techniques that leverage reconfigurable hardware to obtain very precise measurements of the performance of an embedded system on a sample application (BLASTN, as described in Section 2). Section 3 describes the reconfigurable platform that is the vehicle for the research and experiments reported in this paper. In Section 4, we illustrate our approach for collecting statistics by showing how to collect timing information using an FPGA-deployed circuit; here we extend recent work [31] on using reconfigurable architectures for profiling. Our approach does not instru-

*Sponsored by National Science Foundation under grant ITR–0313203.

ment software at all, but relies instead on reconfigurable hardware and replicated logic to monitor the cycles spent in each code segment of interest. In Section 5, we extend this work to measure the performance of an on-chip cache, the characteristics of which are otherwise beyond measurement on a standard processor, except through simulation. We then examine various cache configurations for our particular benchmark, using our platform to conduct expeditiously those experiments necessary for determining a reasonable configuration in terms of footprint and performance. We summarize related work in Section 6 and offer conclusions in Section 7.

2 The BLASTN Algorithm

Basic Local Alignment Search Tool (BLAST) [1] programs are the most widely employed set of software tools for comparing genetic material. They are used to search large databases in computational biology for optimal local alignments to a query. BLASTN ("N" for *nucleotide*) is a variant of BLAST used to compare DNA sequences (lower-level than proteins). A DNA sequence is a string of characters (*bases*), with each base drawn from the 4-symbol alphabet {A,C,T,G}.

BLASTN is classically implemented as a three-stage pipeline, where each stage performs progressively intensive work over a decreasing volume of data.

We analyze the performance of an open-address, double-hashing scheme to determine word matches as in stage 1 of BLASTN. We use a synthetically generated database and query containing only bases from {A, C, T, G}, for our experiments. In our experiments the bases were generated within the program, using random number generators. For the purposes of these experiments, we used a word size (w) of 11, which is also the default value of w used by the flavor of BLASTN that is distributed by the National Center for Biological Information (NCBI).

3 Approach

The Liquid architecture system was implemented as an extensible hardware module on the Field-programmable Port Extender (FPX) platform [25].

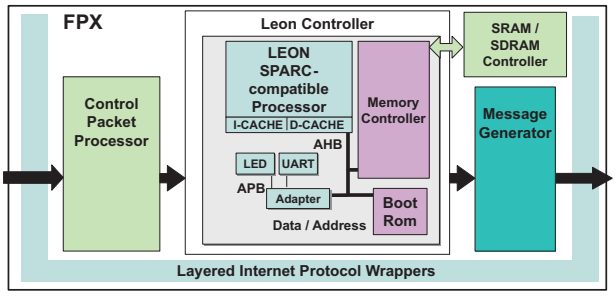

Figure 1: High-level architecture of the Liquid processor system

The major goal of our project is to measure and improve application performance, by providing an easily and efficiently reconfigurable architecture along with software support to expedite its use. We have gathered or developed the components described next.

Liquid Processor Module Figure 1 shows the high level architecture of the Liquid processor module. To interconnect the hardware to the Internet, the logic within the module was fit within the Layered Protocol Wrappers [7]. The wrappers properly format incoming and outgoing data as User Datagram Protocol / Internet Protocol (UDP/IP) network packets. A Control Packet Processor (CPP) routes Internet traffic that contains LEON specific packets (command codes) to the LEON controller (leon_ctrl). The leon_ctrl entity uses these command codes to direct the LEON processor (Restart, Execute) and to read and write the contents of the external memory that the LEON processor uses for instruction and data storage. Finally, the Message Generator is used to send IP packets in response to receiving a subset of the command codes (e.g. Read Memory, LEON status).

LEON Processor System Figure 1 also illustrates some of the main components of the LEON processor base system [14], currently used by the European Space Agency for embedded systems development. As can be seen in Figure 1, the LEON processor system provides fairly sophisticated computer architecture model. It has many features, such as instruction and data caches, the full SPARC V8 instruction

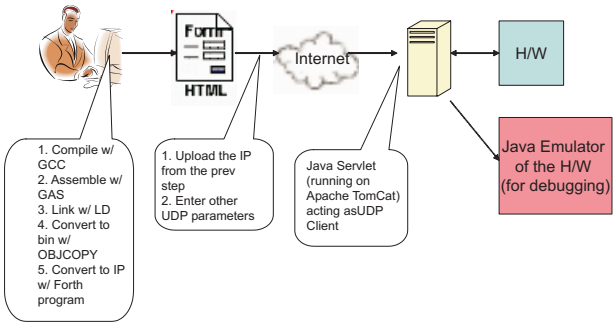

Figure 2: Control software tool chain

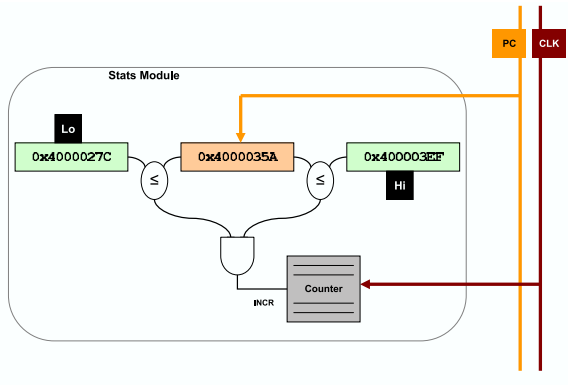

Figure 3: Circuit deployed per-method to profile cycles spent in that method

set, and separate buses for high-speed memory access and low-speed peripheral control.

For inclusion in the Liquid processor system, it was necessary to modify portions of the LEON processor system to interface with the FPX platform. One such modification was to change the Boot ROM, such that the LEON processor begins execution of user code using the FPX platform's RAM.

All other components necessary to implement the interface between the LEON processor system, RAM, and User were implemented outside of the base LEON processor system.

Cross Compiler and Control Software We use LEOX 1.0.18, a LEON toolbox open-source distribution [24]. A memory map is extracted from the C program and supplied to the loader in our batch file.

The web-based control software provides an interface to load compiled instructions over the Internet into LEON's memory. The different components of the control software system are shown in Figure 2.

When users submit a request from the web interface, the request is received by a Java servlet running on an Apache Tomcat server. The servlet creates UDP (IP) control packets and sends them to the Liquid processor module, at a specified destination IP and port. It then waits for a response and handles the display of the response. It is also responsible for error handling during transmission.

4 Cycle-Accurate Profiling

Using the reconfigurable platform, the details of which are presented in Section 3, we have developed a totally nonintrusive, cycle-accurate approach to measure how time is spent in an application. Instead of inserting code into an application to measure time, we deploy logic in hardware that counts the number of clock cycles spent when the program counter is within a specified range. Figure 3 illustrates the circuit that is instantiated for each range of interest:

- The `gcc` cross-compiler emits a load map that shows the addresses at which each method or function is loaded.
- We process the load map into a web page, an example of which is shown in Figure 4 for our BLASTN application.
- The application developer selects the methods to profile. The address ranges are transmitted to the control processor of our platform using IP packets.
- The range registers of the timing circuits are loaded with the methods' address ranges.
- During execution, our statistics module receives the program counter and the system clock. Each timing circuit concurrently manages its counter, incrementing its cycle-count register when the program counter is within its range.
- When the application program terminates, the cycle counts are transmitted back to the application developer over IP.

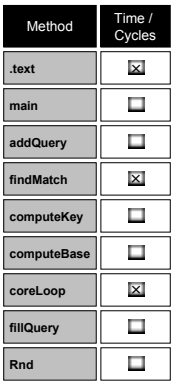

Figure 4: Users select executing-time profiling by method name

The results of our profiling show that most of the execution time is spent in two methods, `coreLoop`, and `findMatch`. While such information is readily obtainable using `gprof`, our approach returns cycle-accurate timing with no application instrumentation.

5 Cache Behavior

The results from Section 4 show that over half of our BLASTN's (word matching) kernel's execution time is spent in the `coreLoop` and `findMatch` methods. Based on our knowledge of this application, we next evaluate the application's performance with regard to the memory-subsystem behavior. In particular, we are concerned about the application's cache performance within the two methods of interest.

While most commercially available processors contain on-chip cache, performance statistics for the cache are not typically exported from the chip. Therefore, it is not possible to obtain cache-performance information on such processors simply by instrumenting the software. Application developers must currently turn to simulation to understand their application's cache behavior; simulation of a processor's cache has the following disadvantages:

- Simulation is much slower than direct execution. As a result, developers must often be satisfied with full simulation of only a small fraction of the program's storage references. Moreover, the simulation typically excludes operating system code and other artifacts that affect a program's performance on an actual system. From our experience using SimpleScalar to obtain detailed performance numbers (such as cache behavior), we have observed more than an order of magnitude increase in completion times.

  Memoization can improve simulation by an order of magnitude [29], and approaches that simulate applications by sampling their phases [32] can offer relief. However, direct execution on the FPGA offers even greater performance and runs an application to completion.

- Simulation is based on an understanding of how the processor's cache behaves under various conditions. For sophisticated cache systems, there can be implementation details that are difficult to replicate accurately in the simulation. Unless the processor's relevant logic is faithfully simulated, the results will be inconsistent with the actual hardware implementation. Moreover, the documentation accompanying processors is often inaccurate concerning the details of the chip's behavior.

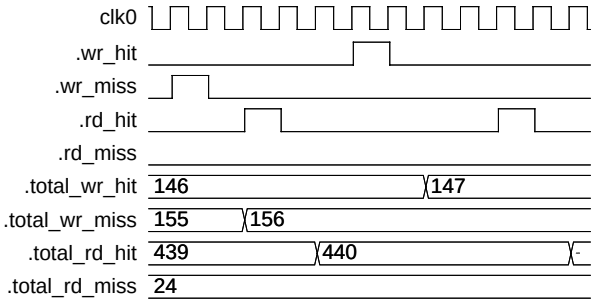

Figure 5: Cache signals extracted from LEON

Owing to the flexibility of our reconfigurable platform, we next describe an approach that measures cache behavior precisely and at full speed. Figure 5 shows signals within LEON that are relevant to the cache subsystem. The top signal is the system clock. The next four signals are the writes (hits and then misses) and the reads (hits and then misses). We obtained these signals by modifying LEON's VHDL to set the signals at appropriate points in the cache

subsystem code. These signals are placed on our Event Monitoring Bus; based on the current program counter, the results are accumulated per-method as described in Section 4. The bottom four signals show the accumulation of counts, based on the four signals above.

5.1 Results for our Benchmark

LEON's cache microarchitecture is parameterized, so that the cache's line size, overall size, and set associativity can be easily selected at configuration time. We next change these parameters for the D- and I-caches and evaluate the cost (in terms of the area occupied by the synthesized bitfile) and performance improvement for the BLASTN benchmark.

To understand how its performance might scale, we varied the size of our benchmark's hash table as follows:

- 32 KB—storing 2,500 11-mers into this hash table
- 128 KB—storing 10,000 11-mers into this hash table

The experiments below are characterized and labeled by the hash-table size.

In its smallest configuration, LEON offers a 1 KB I-cache and a 1 KB D-cache, with both caches mapped directly. From this base configuration, we explore the merits of the following:

- Increasing the I-cache to 4 KB
- Increasing the D-cache to 32 KB
- Increasing the flexibility of mapping into the D-cache to 2-way set associative

Before presenting our performance results, we examine the time taken to obtain these results. A single run, where we searched through a 100 Mbase database to find matches to a 10000 base query, takes 30 mins to evaluate a given cache configuration on our reconfigurable platform, which is currently clocked at 25 MHz. While this clock rate is well below the rate available on today's stock architectures, those architectures cannot provide our results without disturbing at least the program's instruction stream. Moreover, the number of cache misses and hits is not affected by the clock speed.

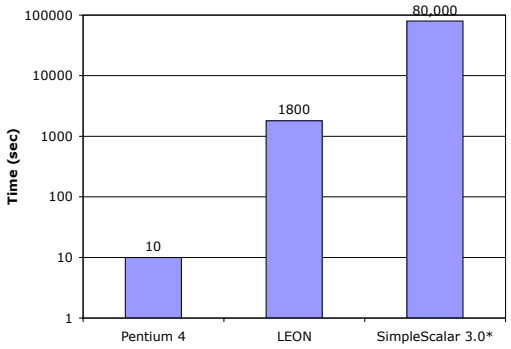

Figure 6: Time to run our application on a Pentium, on our reconfigurable platform, and on a (software) simulator to obtain cache-usage statistics.

Methods	32 KB table		128 KB table	
	Reads	Writes	Reads	Writes
`findMatch`	900	600	900	600
`coreLoop`	1260	426	1261	426
Overall	3295	1526	3289	1527

Figure 7: Benchmark statistics, in millions

If we instead tried to obtain these results through simulation using SimpleScalar, then each run would take nearly a day to complete. Figure 6 [1] illustrates this point quantitatively.

Turning to our benchmark, Figure 7 shows the number of reads and writes issued by our application at each of its hash-table sizes. In all of our experiments, the cache's line size is fixed at 32 bytes; line-size variation is the subject of future work.

For our first set of performance results, Figure 8 shows the D-cache hit rates for various D-cache configurations. While the I-cache is held constant at 1 KB, results for the smallest and largest direct-mapped D-cache are shown. The third row in each set shows a 32 KB, 2-way associative cache. Figure 8

[1] For the Pentium, we used the Pentium 4, 2.6 GHz Processor with 512 KB cache and 768 MB of RAM. The simulation is an extrapolation based on runs using a smaller database of 1 million bases, which completed in ∼13 minutes.

Configuration		Methods		
Size	Assoc	`findMatch`	`coreLoop`	Overall
32 KB Hash Table				
1 KB	1	99.17	91.25	96.27
32 KB	1	99.97	99.90	99.94
32 KB	2	99.99	99.96	99.99
128 KB Hash Table				
1 KB	1	99.18	90.90	96.14
32 KB	1	99.97	93.47	97.69
32 KB	2	99.99	94.52	98.08

Figure 8: Benchmark performance for 1 KB I-cache. D-cache hit rates (percentages) are shown for various D-cache configurations

Configuration		Methods		
Size	Assoc	`findMatch`	`coreLoop`	Overall
32 KB Hash Table				
1 KB	1	3.31	1.00	2.26
32 KB	1	3.33	1.00	2.31
32 KB	2	3.33	1.00	2.31
128 KB Hash Table				
1 KB	1	3.19	1.00	2.19
32 KB	1	3.21	1.00	2.47
32 KB	2	3.21	1.00	2.21

Figure 9: For each D-cache configuration, the *speedup* resulting from expanding the 1 KB I-cache to 4 KB is shown

shows that the increase in D-cache size to 32 KB does not improve the hit rate dramatically for the the 128 KB hash table. However, for the 32 KB hash table, most of the table fits in the D-cache; hence, the number of misses decreases significantly.

While the improvement in D-cache performance is impressive, we also learned from these runs that the data cache is actually *not* crucial to this program's performance. Given the number of storage operations shown in Figure 7 and the relatively high ($\sim$90%) hit rate in the (smallest) D-cache, our application spent on the order of 5% of its execution time in the data storage subsystem.

Further analysis, based on these results, determined that most memory accesses are apparently for stack-allocated variables, which explains the very high hit rates seen in the D-cache. Also, misses in cache occur primarily when accessing the hash table. The later claim is easily verified by comparing the miss rates in the 32 KB D-cache for the 128 KB and 32 KB hash tables.

We next examine the effects of increasing the I-cache from 1 KB to 4 KB. Figure 9 shows the speedup obtained by moving to a 4 KB I-cache for each of the D-cache configurations. While the effect on `coreLoop` is negligible, the performance of `findMatch` is nearly tripled; moreover, the overall program's performance is more than doubled, for each D-cache configuration, by moving to the 4 KB I-cache.

Size	Associativity	Logic	Block Ram
1 KB I-cache			
1 KB	1	45%	28%
32 KB	1	45%	71%
32 KB	2	50%	71%
4 KB I-cache			
1 KB	1	46%	32%
32 KB	1	46%	75%
32 KB	2	51%	75%

Figure 10: Resources consumed by various cache configurations

In summary of our findings, our platform enabled us to determine that the smallest D-cache (1 KB) with the largest I-cache (4 KB) yielded an execution time of 25.77 minutes on the 128 KB hash table. At the other extreme, a 32 KB D-cache and a 1 KB I-cache took 56.13 minutes . The difference in execution time is a *factor* of 2.17 .

We next analyze how the FPGA's resources are best spent, based on the results we show above. Figure 10 compares the area consumed by the configurations we tested. For each configuration, the third column shows the percentage of the FPGA's area that is occupied by the entire synthesis. The fourth column shows the percentage of the FPGA's block RAM that is consumed by all components in the synthesis. Because the cache entries are implemented in block RAM, there is a substantial increase in block RAM usage when the cache is increased from 1 KB to 32 KB. Increasing the set-associativity of the data cache has no impact on the block RAM and only minimal impact ($\sim$5%) on the occupied space.

Based on the performance numbers shown in Figure 8, increase in cache size is strongly correlated with cache performance for benchmark when it uses the smaller hash table. Given the marked increase in block RAM usage, and assuming there may be other uses to which that block RAM could be dedicated, it is possible that a smaller D-cache (of 4 KB

or 8 KB) should be tried with higher set-associativity, since the impact on logic space would be fairly minimal.

When Figure 9 is taken into account, it is clear that the best investment of space in terms of performance is the increased I-cache. The larger I-cache causes a modest increase in the logic space ($\sim$1%), uses $\sim$4% more block RAM, but cuts the runtime by more than half. On the other hand, expansion of the D-cache consumes about the same amount of block RAM but cuts the execution time by only 5%.

6 Related Work

6.1 Heuristic Approaches for Performance Modeling and Measurement

The performance modeling approach that yields estimates quickest is analytical models, often derived directly from the source code. Examples of this include: [6], which describes an approach for the analytical modeling of runtime, idealized to the extent that cache behavior isn't included; and [36], a classic paper on estimating software performance in a codesign environment, which reports accuracy of about $\pm$20%.

Due to the simplifying assumptions that are necessary to yield tractable results, analytic models are notorious for giving inaccurate performance predictions. Moreover, application models often require sophisticated knowledge of the application itself. By contrast, simulation and the direct execution we propose are both "black box" approaches.

The method normally used to improve accuracy beyond modeling is *simulation*. Simulation toolsets commonly used in academia include: SimpleScalar [4], IMPACT [9], and SimOS [28].

Given the long runtimes associated with simulation modeling, it is common practice to limit the simulation execution to only a single application run, not including the OS and its associated performance impact. SimOS does support modeling of the OS, but requires the simulation user manage the time/accuracy tradeoffs inherent in simulating such a large complex system.

Performance monitoring in a relatively non-intrusive manner using hardware mechanisms built into the processor is an idea that is supported on a number of modern systems. Sprunt [34] describes the specific counters, etc. built into the Pentium 4 for exactly this purpose. In an attempt to generalize the availability of these resources in a processor-independent manner, the Performance Application Programmer Interface (PAPI) [8] has been designed to provide a processor independent access path to the types of event counters built into many modern processors. There are a number of practical difficulties to this approach, however, as described in [11]. First, since the underlying hardware mechanisms rely on a software interface, they are not truly non-intrusive. Second, the specific semantics of each mechanism is often not well documented by the manufacturer, even to the point where similarly named items on different systems have subtly different meanings. Finally, there are a number of items of interest, such as cache behavior, that are not available at all via these mechanisms.

Our work is not the first attempt to use the reconfigurable nature of FPGAs to support performance measurements on soft-core processors. Shannon and Chow [31] have developed SnoopP, a non-intrusive snooping profiler. SnoopP supports the measurement of the number of clock cycles spent executing specified code regions. The address ranges that define code regions are specified at synthesis time. This work builds upon that idea, adding the ability to monitor additional features of the system (e.g., cache activity).

Others have used FPGAs as rapid-prototyping platforms to explore various aspects of a processor's design, including its performance. Ray and Hoe [27] have synthesized and executed a superscalar speculative out-of-order core for the integer subset of the SimpleScalar PISA. Simpler systems (more on a par with our approach, complexity-wise) include [16] and [33].

6.2 Customization for Improved Performance

There has been significant work centered around the idea of customizing a processor for a particular application or application set. A few examples are provided here. Arnold and Corporaal [3] describe techniques for compilation given the availability of special function units. Atasu et al. [22] describe the de-

sign of instruction set extensions for flexible ISA systems. Choi et al. [10] examine a constrained application space in their instruction set extensions for DSP systems. Gschwind [17] uses both scientific computations as well as Prolog programs as targets for his instruction set extensions.

Gupta et al. [18] developed a compiler that supports performance-model guided inclusion or exclusion of specific hardware functional units. Systems that use exhaustive search for the exploration of the architecture parameter space are described in [20, 23, 30].

Heuristic design space exploration for application-specific processors is considered in [13]. Pruning techniques are used to diminish the size of the necessary search space in order to find a Pareto-optimal design solution.

In [5], the authors use a combination of analytic performance models and simulation-based performance models to guide the exploration of the design search space. Here, the specific application is in the area of sensor networks. Analytic models are used early, when a large design space is narrowed down to a "managable set of good designs," and simulation-based models are used to provide greater detail on the performance of specific candidate designs.

The AutoTIE system [15] is a development tool from Tensilica, Inc., that assists in the instruction set selection for Tensilica processors. This tool exploits profile data collected from executions of an application on the base instruction set to guide the inclusion or exclusion of candidate new instructions.

7 Conclusion and Future Work

We have presented our results on using a reconfigurable architecture to evaluate some performance metrics—time and cache hits—for one modest application. Our results were obtained from direct execution of the benchmark on a reconfigurable architecture, at full speed and with noninvasive (hardware-deployed) assists for measuring and accumulating the statistics. As future work we shall extend this work to obtain statistics related to other architecture and microarchitecture features, such as instruction counts, branch-table histories, and pipeline activity.

Of course, our approach of adding profiling circuitry can consume increasing area in the FPGA as the number of events and methods of interest grows. We are currently investigating an approach whose area grows with the number of events and methods actually of interest, rather than the larger number of potential combinations of events and methods.

Our evaluation of BLASTN revealed that spending extra chip resources on I-cache was comparably much more effective than spending those same resources expanding or increasing the set-associativity of the D-cache. Our evaluation sampled only a fraction of the microarchitecture configurations that could be deployed and evaluated. In future work, we seek to explore and identify efficient configurations automatically. Generation of all possible combinations is prohibitive; thus, research is needed to explore the microarchitecture design space intelligently.

As we move from the microarchitecture to the ISA, we are interested in identifying structures that improve the performance of specific applications, such as software pipelines, new instructions, and specialized data types.

Acknowledgements

We thank James Brodman of the DOC group for his help preparing the figures for this paper. We thank Ed Kotysh and Paul Spiegel for their careful proofreading.

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] ARC International. `http://www.arccores.com`.

[3] Marnix Arnold and Henk Corporaal. Designing domain-specific processors. In *Proc. of the 9th Int'l Symp. on Hardware/Software Codesign*, pages 61–66, April 2001.

[4] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer sys-

tem modeling. *IEEE Computer*, 35(2):59–67, February 2002.

[5] Amol Bakshi, Jingzhao Ou, and Viktor K. Prasanna. Towards automatic synthesis of a class of application-specific sensor networks. In *Proc. of Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 50–58, 2002.

[6] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto. Source-level execution time estimation of C programs. In *Proc. of the 9th Int'l Symp. on Hardware/Software Codesign*, pages 98–103, April 2001.

[7] Florian Braun, John Lockwood, and Marcel Waldvogel. Protocol wrappers for layered network packet processing in reconfigurable hardware. *IEEE Micro*, 22(3):66–74, January 2002.

[8] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int'l Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[9] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu. IMPACT: an architectural framework for multiple-instruction-issue processors. In *Proc. of the 18th Int'l Symp. on Computer Architecture*, May 1991.

[10] Hoon Choi, Jong-Sun Kim, Chi-Won Yoon, In-Cheol Park, Seung Ho Hwang, and Chong-Min Kyung. Synthesis of application specific instructions for embedded DSP software. *IEEE Trans. on Computers*, 48(6):603–614, June 1999.

[11] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou. Experiences and lessons learned with a portable interface to hardware performance counters. In *Proc. of Workshop on Parallel and Distributed Systems: Testing and Debugging (at IPDPS)*, April 2003.

[12] J. E. Carrillo Esparza and P. Chow. The effect of reconfigurable units in superscalar processors. In *Proc. ACM Int'l Symp. on Field Programmable Gate Arrays*, pages 141–150, 2001.

[13] Dirk Fischer, Jürgen Teich, Michael Thies, and Ralph Weper. Efficient architecture/compiler co-exploration for asips. In *Proc. of Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 27–34, 2002.

[14] Gaisler Research. `http://www.gaisler-.com`.

[15] David Goodwin and Darin Petkov. Automatic generation of application specific processors. In *Proc. of Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 137–147, 2003.

[16] M. Gschwind, V. Salapura, and D. Maurer. FPGA prototyping of a RISC processor core for embedded applications. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 9(2):241–250, April 2001.

[17] Michael Gschwind. Instruction set selection for ASIP design. In *Proc. of the 7th Int'l Symp. on Hardware/Software Codesign*, pages 7–11, May 1999.

[18] T. Vinod Kumar Gupta, Roberto E. Ko, and Rajeev Barua. Compiler-directed customization of ASIP cores. In *Proc. of the 10th Int'l Symp. on Hardware/Software Codesign*, pages 97–102, May 2002.

[19] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The Chimaera reconfigurable functional unit. In *Proc. IEEE Symp. on FPGAs for Custom Computing Machines*, pages 87–96, 1997.

[20] Olivier Hebert and Yvon Savaria Ivan C. Kraljic. A method to derive application-specific embedded processing cores. In *Proc. of the 8th Int'l Symp. on Hardware/Software Codesign*, pages 88–92, May 2000.

[21] Phillip Jones, Shobana Padmanabhan, Daniel Rymarz, John Maschmeyer, David V. Schuehler, John W. Lockwood, and Ron K. Cytron. Liquid architecture. In *Workshop on Next Generation Software (at IPDPS)*, 2004.

[22] Paolo Ienne Kubilay Atasu, Laura Pozzi. Automatic application-specific instruction-set extensions under microarchitectural constraints.

In *Proc. of Design Automation Conf.*, June 2003.

[23] Mika Kuulusa, Jari Nurmi, Janne Takala, Pasi Ojala, and Henrik Herranen. A flexible DSP core for embedded systems. *IEEE Design and Test of Computers*, 14(4):60–68, 1997.

[24] LEOX.org. `http://www.leox.org`.

[25] John W. Lockwood. The Field-programmable Port Extender (FPX). `http://www.arl.wustl.edu/arl/-projects/fpx/`, December 2003.

[26] Christian Plessl, Rolf Enzler, Herbert Walder, Jan Beutel, Marco Platzner, Lothar Thiele, and Gerhard Troester. The case for reconfigurable hardware in wearable computing. *Personal and Ubiquitous Computing*, 7(5):299–308, 2003.

[27] Joydeep Ray and James C. Hoe. High-level modeling and FPGA prototyping of microprocessors. In *Proc. ACM Int'l Symp. on Field Programmable Gate Arrays*, pages 100–107, February 2003.

[28] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Trans. on Modeling and Computer Simulation*, 7(1):78–103, January 1997.

[29] Eric Schnarr and James R. Larus. Fast out-of-order processor simulation using memoization. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 283–294. ACM Press, 1998.

[30] Barry Shackleford, Mitsuhiro Yasuda, Etsuko Okushi, Hisao Koizumi, Hiroyuki Tomiyama, and Hiroto Yasuura. Memory-CPU size optimization for embedded system designs. In *Proc. of Design Automation Conf.*, pages 246–251, June 1997.

[31] Lesley Shannon and Paul Chow. Using reconfigurability to achieve real-time profiling for hardware/software codesign. In *Proc. ACM Int'l Symp. on Field Programmable Gate Arrays*, pages 190–199, 2004.

[32] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57. ACM Press, 2002.

[33] Kyung soo Oh, Sang yong Yoon, and Soo-Ik Chae. Emulator environment based on an FPGA prototyping board. In *Proc. of 11th IEEE Int'l Workshop on Rapid System Prototyping*, pages 72–77, June 2000.

[34] Brinkley Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82, 2002.

[35] Stretch, Inc. `http://www.stretchinc-.com`.

[36] Kei Suzuki and Alberto Sangiovanni-Vincentelli. Efficient software performance estimation methods for hardware/software codesign. In *Proc. of Design Automation Conf.*, pages 605–610, June 1996.

[37] Tensilica, Inc. `http://www.tensilica-.com`.