THE VECTOR-THREAD ARCHITECTURE

THE VECTOR-THREAD (VT) ARCHITECTURE SUPPORTS A SEAMLESS INTERMINGLING OF VECTOR AND MULTITHREADED COMPUTATION TO FLEXIBLY AND COMPACTLY ENCODE APPLICATION PARALLELISM AND LOCALITY. VT PROCESSORS EXPLOIT THIS ENCODING TO PROVIDE HIGH PERFORMANCE WITH LOW POWER AND SMALL AREA.

Ronny Krashinsky
Christopher Batten
Mark Hampton
Steve Gerding
Brian Pharris
Jared Casper
Krste Asanović
Massachusetts Institute of Technology

•••••• As embedded computing applications become more sophisticated, the demand for high-performance, low-power information processing grows. Custom circuits provide the optimal solution for any processing task, but rising mask and development costs limit application-specific chips to devices that will be sold in very high volumes. Even then, hardwired circuits are unsuitable for designs that must rapidly adapt to changing requirements. Programmable domain-specific processors are a more flexible alternative, and they have evolved to exploit particular forms of parallelism common to certain classes of embedded applications. Examples include digital signal processors, media processors, network processors, and field-programmable gate arrays. Full systems, however, often require a heterogeneous mix of these cores to be competitive on complex workloads with a variety of processing tasks. The resulting devices have multiple instruction sets with different parallel execution and synchronization models, making them difficult to program. They are also inefficient when the application workload causes load imbalance across the heterogeneous cores.

Ideally, a single all-purpose programmable architecture would efficiently exploit the different types of parallelism and locality present in embedded applications. Although general-purpose processors are flexible, they are often too large, too slow, or burn too much power for embedded computing. In particular, modern superscalars expend considerable hardware resources to dynamically extract limited parallelism from sequential encodings of unstructured applications. All-purpose processors have a different goal. Embedded applications often contain abundant structured parallelism, where dependencies can be determined statically. The challenge is to develop an instruction set architecture that flexibly encodes parallel dependency graphs and improves the performance-efficiency of processor implementations.

Vector-thread architectural paradigm

The vector-thread (VT) architectural paradigm describes a class of architectures that unify the vector and multithreaded execution models. VT architectures compactly encode large amounts of structured parallelism in a form that lets simple microarchitectures attain high performance at low power by avoiding complex control and datapath structures and by reducing activity on long wires.

Abstract model

A VT programming model combines vector and multithreaded computation. A con-

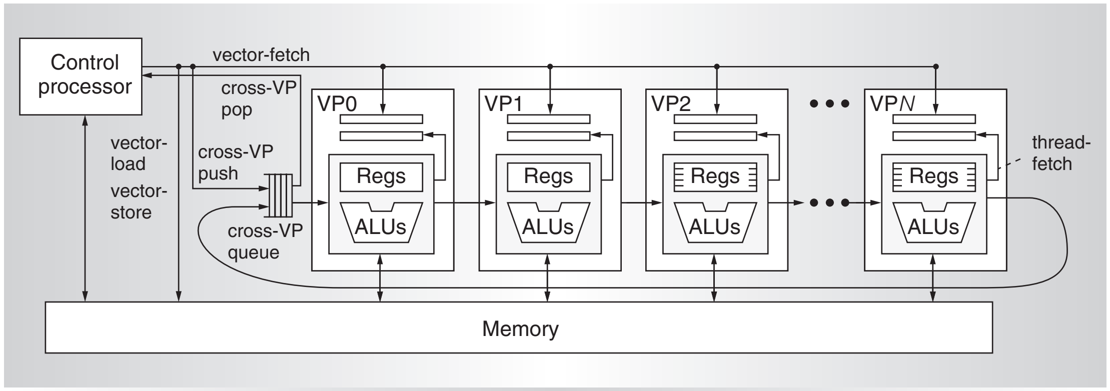

Figure 1. Abstract model of a vector-thread architecture. A control processor interacts with a vector of virtual processors (VPs).

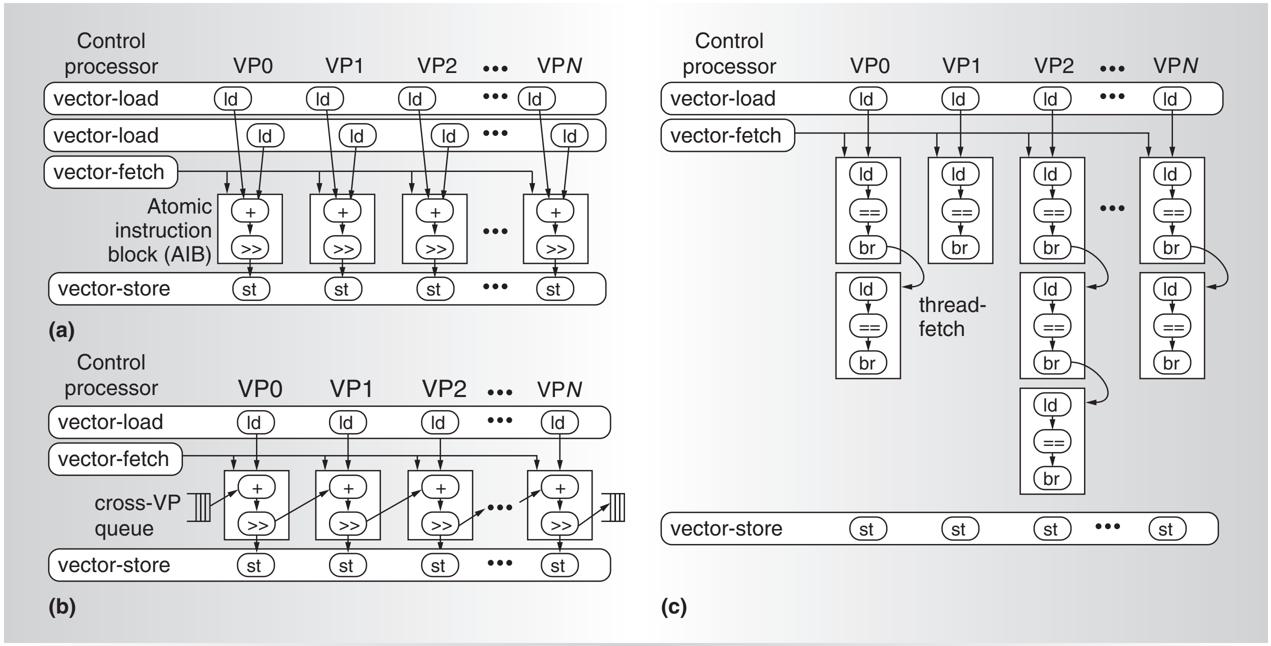

Figure 2. Mapping code to the vector-thread architecture: vectorizable loops (a), loops with cross-iteration dependencies (b), and loops with internal control flow (c).

ventional control processor interacts with a vector of virtual processors (VPs), as Figure 1 shows. A virtual processor contains a set of registers and can execute groups of RISC-like instructions packaged into atomic instruction blocks (AIBs). VPs have no automatic program counter or implicit instruction fetch mechanism; rather, all instruction blocks must be explicitly requested by either the control processor or the VP itself.

Although we can map applications to VT in several ways, VT is especially well suited to executing loops. Each VP executes a single loop iteration, and the control processor strip-mines the execution and factors out common bookkeeping overhead. To execute data-parallel code, the control processor uses vector-fetch commands to broadcast AIBs to all VPs, as Figure 2a shows. The control processor can also use vector-load and vector-store commands to efficiently move vectors of data between memory and the VP registers. To allow efficient mapping of loop-carried dependencies to VT, VPs are connected in a unidirectional ring topology, with pairs of sending and receiving instructions transferring data

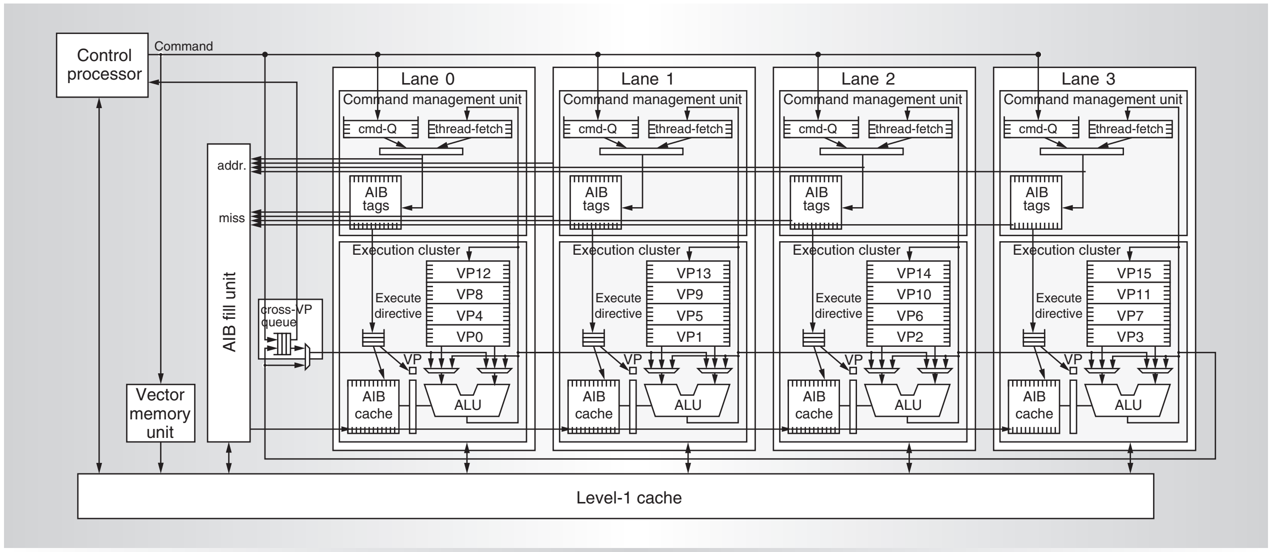

Figure 3. Physical model of a VT machine. The implementation shown has four parallel lanes in the vector-thread unit (VTU), and VPs are striped across the lane array.

directly between them, as Figure 2b illustrates. In contrast to software pipelining on VLIW architectures, the compiler or programmer need only schedule code for one loop iteration mapped to one VP, and the hardware dynamically schedules the cross-VP data transfers to resolve when the data becomes available.

To execute loop iterations with conditionals or even inner loops, each VP directs its own control flow using thread fetches to fetch its AIBs, as Figure 2c shows. By allowing software to freely intermingle vector and thread fetches, a VT architecture can combine the best attributes of the vector and multithreaded execution paradigms. The control processor can issue a vector-fetch command to launch a vector of VP threads, each of which continues to execute as long as the VP issues thread fetches. Thread fetches break the rigid control flow of traditional vector machines, letting the VPs follow independent control paths. By letting VPs conditionally branch, VT provides more efficient execution of large conditionals than traditional vector masking. Thread fetches also let outer-loop parallelism map naturally to VT, as VPs can execute inner-loop iterations that have little or no available parallelism. In addition to executing loop iterations, VPs can serve as free-running threads, operating independently from the control processor and retrieving tasks from a shared work queue.

Physical model

A VT machine contains a conventional control processor and a vector-thread unit (VTU) that executes the VP code, as Figure 3 illustrates. To exploit the parallelism exposed by the VT abstract model, the VTU contains a parallel array of processing lanes. Each lane contains physical registers, which hold the state of VPs mapped to the lane, and functional units, which are time-multiplexed across the VPs.

Unlike traditional vector machines, in a VT machine the lanes execute decoupled from each other, and each lane has a small AIB cache. A lane's command management unit (CMU) buffers commands from the control processor in a queue (cmd-Q) and holds pending thread-fetch addresses for the lane's VPs. The CMU chooses a vector-fetch or thread-fetch command to process, and looks up its address in the AIB cache tags. After processing an AIB cache hit or miss refill, the CMU generates an execute directive containing an index to the AIB cache. For a vector-fetch command, the execute directive indicates that all VPs should execute the AIB; for a thread-fetch command, it identifies a single VP to execute the AIB. The CMU sends execute directives to a queue, and it can overlap the AIB cache refill for new fetch commands with the execution of previous ones.

To process an execute directive, the execution cluster reads VP instructions one by one from the AIB cache and executes them for the appropriate VP. When processing an execute directive from a vector-fetch command, the execution cluster executes all of the instructions in the AIB for one VP before moving on to the next.

Scale VT processor

The Scale architecture, an instantiation of the VT paradigm, aims to provide high performance at low power for a wide range of embedded applications while using only a small area. The planned Scale prototype includes a MIPS-RISC control processor, 32 Kbytes of cache, and a four-lane vector-thread unit that can execute 16 operations per cycle and supports up to 128 simultaneously active virtual processor threads. The estimated area is only 10 mm^2 in 0.18-μm technology. Here, we give a brief overview of Scale; a more detailed description and evaluation is available elsewhere.[1,2]

To optimize area and energy, Scale partitions lanes (and VPs) into multiple execution clusters, each containing only a subset of all possible functional units and a small register file with few ports. The atomic execution of AIBs lets Scale expose shared temporary state that is valid only within an AIB—chain registers at each ALU input reduce register file energy, and shared VP registers reduce the register file size needed to support a large number of VPs. Each cluster has independent control, and intercluster register dependencies are statically partitioned into transport and writeback micro-ops, enabling decoupled cluster execution (without requiring dynamic register renaming[3]). Additionally, the memory access cluster uses load-data and store-address queues to enable access/execute decoupling.[4] Decoupling allows each in-order cluster to execute code for different VPs on the same cycle, providing an inexpensive form of simultaneous multithreading to hide large functional unit or memory latencies.

The Scale memory system uses cache refill/access decoupling[2] to pre-execute vector loads and issue any needed cache line refills before regular execution. Scale also provides segment vector loads and stores, which efficiently pack and unpack multifield records into VP registers. These mechanisms let Scale use a conventional cache to amplify memory bandwidth and tolerate long memory latencies, and avoid complicating the software interface with a hierarchical vector (or stream) register file.[5,6]

To evaluate Scale's performance and flexibility, we mapped a diverse selection of embedded benchmarks (including examples from cryptography and image, audio, and network processing) from the Embedded Microprocessor Benchmark Consortium (EEMBC) and other suites. The results in Tables 1 and 2 are based on a detailed execution-driven simulator,[1] and they represent a snapshot of our ongoing progress in optimizing the microarchitecture and benchmark mappings. The simulator modeled a four-lane Scale configuration running at 400 MHz with a 64-bit wide DDR2 memory interface clocked at 200 MHz. As is standard practice for EEMBC, we give results for compiled out-of-the-box code (OTB), which runs on the Scale control processor, and assembly-optimized code (OPT), which makes use of the VTU. Total cycle numbers for non-EEMBC benchmarks (Table 2) are for the entire application, while the remaining statistics are for the kernel only (the kernel excludes benchmark overhead code and for Li the kernel consists of the garbage collector only). Results for different data sets appear separately, or an "All" data set indicates that results were similar across inputs. Overall, the results show that Scale can flexibly provide competitive performance on a wide range of codes from different domains.

Advantages

VT draws from earlier vector architectures,[7] and like vector microprocessors,[8-10] the Scale VT implementation provides high throughput at low complexity. Vector-fetch commands issue many parallel instructions simultaneously, while vector-load and vector-store commands encode data locality and allow optimized memory access. A vector-fetch broadcasts an AIB address to all lanes, which each perform only a single tag check. The execution cluster then reads instructions within the AIB using a short index into the small AIB cache, and the vector-fetch ensures that each VP in a lane will reuse the AIB before any eviction is possible. Vector-memory commands improve performance and save

Table 1. Performance and mapping characterization for benchmarks from the Embedded Microprocessor Benchmark Consortium (EEMBC).

			Iterations per second			Per-cycle statistics					Memory
Benchmark	Description	Data set	OTB	OPT	Kernel speedup	Ops	Ld-El	St-El	Mem-B	Loop types	Memory access types
Rgbcmy	RGB to CMYK color conversion	—	140	2,080	14.8	6.8	1.2	0.4	3.0	DP	VM, SVM
Rgbyiq	RGB to YIQ color conversion	—	58	2,308	39.8	9.3	1.3	1.3	3.8	DP	SVM
Rgbhpg	High pass gray-scale filter	—	111	4,962	44.6	10.4	2.8	1.0	3.1	DP	VM, VP
Text	Printer language parsing	—	298	440	1.5	0.2	0.0	0.0	0.0	DE	VM
Dither	Floyd-Steinberg gray-scale dithering	—	149	1,027	6.9	5.0	1.1	0.3	0.3	DP, DC	VM, SVM, VP
Rotate	Binary image 90-degree rotation	—	707	17,948	25.4	11.0	0.6	0.6	0.0	DP	VM, SVM
Lookup	IP route lookup using Patricia Trie	—	1,662	9,661	5.8	6.9	0.9	0.0	0.0	DI	VM, VP
Ospf	Djikstra shortest path first	—	6,197	6,995	1.1	1.3	0.2	0.1	0.1	FT	VP
Pktflow	IP packet processing	512 Kbytes	6,637	124,032	18.7	7.5	1.5	0.1	0.6	DC, XI	VM, VP
		1 Mbyte	2,362	31,260	13.2	3.7	0.7	0.1	4.2		
		2 Mbytes	1,206	16,274	13.5	3.7	0.7	0.1	4.2		
Pntrch	Pointer chasing, searching linked list	—	8,828	38,639	4.4	2.3	0.3	0.0	0.0	FT	VP
Fir	Finite impulse response filter	—	57,362	6,331,117	110.4	6.9	1.7	0.1	0.5	DP	VM, SVM
Fbital	Bit allocation for DSL modems	Typ	860	22,767	26.5	3.6	0.5	0.2	0.0	DC, XI	VM, VP
		Step	12,533	286,944	22.9	2.3	0.4	0.0	0.0		
		Pent	1,304	61,387	47.1	3.4	0.5	0.0	0.0		
Fft	256-pt fixed-point complex FFT	All	6,577	123,959	18.8	3.8	1.7	1.4	0.1	DP	VM, SVM
Viterb	Soft decision Viterbi decoder	All	1,561	16,316	10.5	5.0	0.5	0.5	0.1	DP	VM, SVM
Autocor	Fixed-point autocorrelation	Data1	280,259	3,208,985	11.5	3.2	1.2	0.1	0.1	DP	VM
		Data2	1,889	64,143	34.0	7.9	2.8	0.0	0.0		
		Data3	1,980	79,000	39.9	9.5	3.3	0.0	0.0		
Conven	Convolutional encoder	Data1	2,858	2,676,480	936.4	10.3	0.9	0.2	0.9	DP	VM, VP
		Data2	3,320	3,476,749	1,047.1	11.1	1.1	0.3	0.9		
		Data3	4,216	4,294,149	1,018.4	10.3	1.4	0.3	1.4		

Note: The abbreviations in the table, as well as in Table 2, are: out-of-the-box code (OTB), assembly-optimized code (OPT), VTU compute operations (Ops), load elements (Ld-El), store elements (St-El), and DRAM bytes (Mem-B). Loop types parallelized in the Scale mappings include: data-parallel loop with no control flow (DP), data-parallel loop with conditional thread fetches (DC), loop with cross-iteration dependencies (XI), data-parallel loop with inner-loop (DI), loop with data-dependent exit condition (DE), and free-running threads (FT). Memory access types include: unit-stride and strided vector memory accesses (VM), segment vector memory accesses (SVM), and individual VP loads and stores (VP).

memory-system energy by avoiding the additional arbitration, tag checks, and bank conflicts that would occur if each VP requested elements individually.

In executing loops with cross-iteration dependencies, VT has many advantages over VLIW architectures. Loop iterations map to VPs on parallel lanes, and the dynamic hardware scheduling of explicit cross-VP data transfers allows execution to automatically adapt to the software critical path. Furthermore, within a lane, Scale's cluster decoupling lets execution dynamically adjust to functional unit and memory latencies. These features alleviate the need for loop unrolling, software pipelining, and static scheduling, enabling more portable and compact code.

VT provides extremely fine-grained multithreading with low overhead; a single vector-fetch command can launch 100 threads, each executing 10 instructions. This granularity lets VT parallelize code more effectively than conventional simultaneous multithreading (SMT) and chip multiprocessor (CMP) architectures. VT factors out bookkeeping code to the control thread, and vector-fetch and vector-memory commands efficiently distribute instructions and data to the VP threads. Additionally, vector fetches and the cross-VP network provide low-overhead fine-grained

Table 2. Performance and mapping characterization for benchmarks from other suites.

Benchmark	Suite	Description	Data set	Total cycles (millions) OTB	Total cycles (millions) OPT	Kernel speedup	Per-cycle statistics Ops	Ld-El	St-El	Mem-B	Loop types	Memory access types
Rijndael_dec	MiBench	AES decryption	Large	412.7	204.2	2.6	2.5	0.8	0.1	0.0	DP	VM, VP
Sha	MiBench	Secure hash algorithm	Large	141.0	59.1	2.4	1.9	0.3	0.1	0.0	DP, XI	VM, VP
Qsort	MiBench	Quick sort of strings	Small	34.2	21.8	3.0	2.0	0.4	0.3	2.2	FT	VP
Adpcm_enc	Mediabench	Speech encoding	—	7.6	4.2	1.9	2.3	0.1	0.0	0.0	XI	VM, VP
Adpcm_dec	Mediabench	Speech decoding	—	6.3	0.9	8.1	6.7	0.6	0.2	0.0	XI	VM, VP
Li	SpecInt95	Lisp interpreter	Test	1,458.6	1,155.0	5.1	2.5	0.3	0.2	2.7	DP, DC, XI, DI	VM, VP

synchronization, and a shared first-level cache enables low-overhead memory coherence.

VT seamlessly combines vector and threaded execution. Unlike polymorphic architectures, which must explicitly switch modes,[11,12] Scale can exploit fine-grained data-, thread-, and instruction-level parallelism simultaneously. Although some architectures can extract fine-grained parallelism from a wider range of loops, VT handles many common parallel loop types while avoiding heavyweight interthread synchronization on shared global registers[13] and speculative execution with dynamic checks for memory dependencies.[14] In comparison to exposed architectures,[6,15] VT provides a high-level virtual processor abstraction, so software can encode parallelism without exposing machine details such as the number of physical registers and processing units.

Conclusion

VT exploits fine-grained parallelism and locality more effectively than traditional superscalar, VLIW, or multithreaded architectures. VT's flexibility enables new ways of parallelizing codes—for example, by letting vector-memory commands feed directly into threaded code. The Scale prototype demonstrates that VT is well-suited to all-purpose embedded computing, letting a single compact design provide competitive performance across a range of applications. In the future, we expect that the vector-thread paradigm will prove to be widely applicable across computing domains. MICRO

Acknowledgments

This work was funded in part by DARPA PAC/C award F30602-00-2-0562, US National Science Foundation CAREER award CCR-0093354, an NSF graduate fellowship, the Cambridge-MIT Institute, donations from Infineon Corporation, and an equipment donation from Intel Corporation.

References

1. R. Krashinsky et al., "The Vector-Thread Architecture," *Proc. Int'l Symp. Computer Architecture* (ISCA-31), IEEE CS Press, 2004, pp. 52-63.
2. C. Batten et al., "Cache Refill/Access Decoupling for Vector Machines," *Proc. 37th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (MICRO-37), IEEE CS Press, 2004, pp. 331-342.
3. C. Kozyrakis and D. Patterson, "Overcoming the Limitations of Conventional Vector Processors," *Proc. Int'l Symp. Computer Architecture* (ISCA-30), ACM Press, 2003, pp. 399-409.
4. J.E. Smith, "Dynamic Instruction Scheduling and the Astronautics ZS-1," *Computer*, vol. 22, no. 7, July 1989, pp. 21-35.
5. T. Watanabe, "Architecture and Performance of NEC Supercomputer SX System," *Parallel Computing*, vol. 5, 1987, pp. 247-255.
6. S. Rixner et al., "A Bandwidth-Efficient Architecture for Media Processing," *Proc. 31st Ann. IEEE/ACM Int'l Symp. Microarchitecture,* (MICRO-31), IEEE CS Press, 1998, pp. 3-13.
7. R.M. Russel, "The CRAY-1 Computer System," *Comm. ACM*, vol. 21, no. 1, Jan. 1978, pp. 63-72.
8. J. Wawrzynek et al., "Spert-II: A Vector Microprocessor System," *Computer*, vol. 29, no. 3, Mar. 1996, pp. 79-86.
9. C. Kozyrakis et al., "Scalable Processors in

the Billion-Transistor Era: IRAM," *Computer*, vol. 30, no. 9, Sept. 1997, pp. 75-78.
10. K. Kitagawa et al., "A Hardware Overview of SX-6 and SX-7 Supercomputer," *NEC Research & Development J.*, vol. 44, no. 1, Jan. 2003, pp. 2-7.
11. K. Mai et al., "Smart Memories: A Modular Reconfigurable Architecture," *Proc. Int'l Symp. Computer Architecture* (ISCA-27), IEEE CS Press, 2000, pp. 161-171.
12. K. Sankaralingam et al., "Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture," *Proc. Int'l Symp. Computer Architecture* (ISCA-30), ACM Press, 2003, pp. 422-433.
13. C.R. Jesshope, "Implementing an Efficient Vector Instruction Set in a Chip Multiprocessor Using Micro-Threaded Pipelines," *Australia Computer Science Comm.*, vol. 23, no. 4, 2001, pp. 80-88.
14. G.S. Sohi, S.E. Breach, and T.N. Vijaykumar, "Multiscalar Processors," *Proc. Int'l Symp. Computer Architecture* (ISCA-22), ACM Press, 1995, pp. 414-425.
15. E. Waingold et al., "Baring It All to Software: Raw Machines," *Computer*, vol. 30, no. 9, Sept. 1997, pp. 86-93.

Ronny Krashinsky is a PhD student in electrical engineering and computer science at MIT. His research interests include high-performance, energy-efficient computer architecture and VLSI design. Krashinsky has a BS from UC Berkeley and an SM from MIT, both in electrical engineering and computer science.

Christopher Batten is a PhD candidate in the MIT Department of Electrical Engineering and Computer Science. His research interests include low-power, high-performance computer architectures and synthetic biological systems. Batten has a BS in electrical engineering from the University of Virginia, and a MPhil in engineering from the University of Cambridge.

Mark Hampton is a PhD student in electrical engineering and computer science at MIT. He has BS degrees in computer and systems engineering as well as computer science from Rensselaer Polytechnic Institute and an SM degree in electrical engineering and computer science from MIT.

Steve Gerding is a graduate student in electrical engineering and computer science at MIT. His research interests include energy and performance measurement of high-performance microprocessors. Gerding has a BS in computer science and engineering from UCLA.

Brian Pharris is currently an employee at NVIDIA. He has an MEng in electrical engineering and computer science from MIT.

Jared Casper is an undergraduate student in electrical engineering and computer science at MIT. His research interests are in extreme computer architectures.

Krste Asanović is an associate professor in the MIT Department of Electrical Engineering and Computer Science and a member of the MIT Computer Science and Artificial Intelligence Laboratory. His research interests are computer architecture and VLSI design. He has a PhD in computer science from the University of California, Berkeley. He is a member of the IEEE and the ACM.

Direct questions and comments to Ronny Krashinsky, MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar St., #32-G736, Cambridge, MA 02139; ronny@mit.edu.