

An Analysis of Research Accelerator for Multiple Processors (RAMP) Gold

Revarr Johnson, Andrew Waterman, and
Krste Asanovic

Email : {rmjohnso, waterman,
krste}@eecs.berkeley.edu

Abstract : Concerning the transition to multicore microprocessing, we argue that FPGA Architecture Model Execution (FAME) simulators can increase the number of useful architecture research experiments per day by over a factor of 1000 versus Software Architecture Model Execution (SAME) simulators. To validate this claim, studies are made on RAMP Gold, a FAME simulator focusing on the speed up curve and scalability as the numbers of cores are increased.

Computer architects have long used software simulators to explore instruction set architectures, microarchitectures, and approaches to implementation. Compared to hardware prototyping, their low capital cost, relatively low-cost of implementation and ease of change have made them the ideal choice in the early stages of research exploration. Architects can explore many variations of a design simply by changing software simulator parameters, and in the era when uniprocessor performance was doubling every 18 months, simulation speed correspondingly doubled every 18 months without any special programming effort. The recent abrupt transition to multicore architectures [1], however, has both increased the complexity of the systems architects want to simulate and removed the straightforward path to simulator performance scaling.

This paper uses the information from the draft of the A Case for FPGA Architecture Model Execution (FAME) paper [2] that is in progress by members of the Par Lab to explain prototyping and the two paths forward in multicore simulation, Software Architecture Model Execution (SAME) and FPGA (Field-Programmable Gate Array) Architecture Model Execution (FAME). While SAME certainly merits continued investigation, we and others in the Research Accelerator for Multiple Processors (RAMP) project are focused

on the potential of FAME via RAMP Gold.

Prototyping

We begin by briefly reviewing the role of prototyping and how prototyping is different from simulation. Hardware prototyping has a long history, reaching back to the very first computers built in universities, such as the Harvard Mark-I and EDSAC, but is much less common today. In the 1980s, many researchers would build prototype chips to illustrate the value of their architectural innovations. For example, the case for RISC architectures was substantially strengthened by the prototype RISC chips built at Berkeley [3] and Stanford [4], which ran programs faster than commercial machines despite being produced by small academic teams. Similarly, the later Stanford DASH [5] and MIT Alewife [6] projects provided considerable insight into the implementation and viability of large-scale directory-based cache-coherent shared memory architectures.

The goals of a research prototype are quite different from that of a simulator. The process of constructing a working prototype is usually far more valuable than the end result, helping inventors understand how a new architectural idea can be implemented. When completed, a successful prototype will provide a credible proof-of-concept to help explain the idea and convince practitioners to adopt the technology. Although prototypes are sometimes justified as a way to gather evaluation data on much larger and longer programs than possible with software simulators, usually this is (or becomes) a secondary concern. In our experience, few computer architecture prototypes support extensive parameterization and instrumentation due to the additional design effort and resources required. Even when hardware hooks are added, the supporting software infrastructure to exploit these features rarely materializes, either because of limited project resources or because experimental needs were not well understood or have changed from when the prototype design was frozen. Prototyping projects also tend to focus implementation effort on the novel mechanisms and often make expedient simplifications in other well-understood areas (e.g., by omitting floating-point hardware or virtual memory), which limits the kinds of software that can be run.

For these reasons, prototyping is not an alternative to simulation. Conversely, when the implementation of a mechanism is not well understood, it is by definition not possible to construct accurate simulator models. Hence, simulation and prototyping are complementary techniques. Early high-level simulations can be used to quickly explore interesting

architectural concepts, and determine which are worthy of further study. Prototyping can then be undertaken for ideas whose implementation is not well understood. A final prototype can provide a proof-of-concept that a particular mechanism is implementable with reasonable efficiency for at least one design point, while a detailed simulator calibrated with implementation knowledge gained from the prototype can accurately model the performance of the proposed mechanism across a much wider variety of target system parameters and with a much fuller feature set than possible with prototypes. Another important difference is that while prototypes are expensive to manufacture and distribute, a calibrated software simulator model can be easily shared with colleagues.

Shrinking feature sizes have led to multi-gigahertz microprocessor clock rates and a rapid growth in architectural complexity. Consequently, the engineering skill, design effort, and fabrication cost required to build a compelling prototype microprocessor have risen to the point where few researchers now contemplate such a project. Even when research prototypes are successfully completed, the quality of implementation is often markedly inferior to production designs, leading to doubts about the relevance of the prototype. But it is primarily the act of prototyping that yields valuable implementation insights, and these are often largely independent of implementation technology. Simulator models can be constructed using the prototype design as a guide, but then extrapolating to model the effect of a more realistic implementation or an advanced future technology.

Given that most of the implementation insights are developed during the prototype design phase and that most of the cost is incurred during fabrication, one viable intermediate approach is to complete a detailed design using VLSI CAD tools without proceeding to fabrication. Simulator models can then be calibrated using timing, area, and power data extracted from the tools. This approach still requires significant engineering skill and design effort using a large and complex tool set, but is ultimately not as credible as a working prototype. Given the high cost and long turnaround of prototype chip fabrication, some researchers are using FPGAs to construct relatively inexpensive and malleable working prototypes. The timing, area, and power of an FPGA prototype are very different from a production chip implementation, but the hardware design process is similar enough to yield many of the same important insights. A promising direction is to combine FPGA prototyping with detailed VLSI CAD design to provide both a

working prototype and believable implementation metrics. While we believe FPGA prototyping can be a valuable tool, the subject of this paper is a very different technology: Using FPGAs to accelerate the execution of highly parameterized and thoroughly instrumented architecture simulators. We have found the difference between FPGA prototypes and FPGA simulators to be one of the main sources of confusion when discussing RAMP with those outside the project, and our goal in this paper is to clarify the difference and provide taxonomy of FPGA simulation approaches.

The Multicore Revolution

As has been widely reported, the end of ideal technology scaling together with the practical power limit for an aircooled chip package forced all microprocessor manufacturers to switch to multiple processors per chip [1]. The path to more performance for such *multicore* designs is increasing the number of *cores* per chip every technology generation, with the cores themselves essentially going no faster.

There are three subtle impacts of the multicore era. First, simulators will no longer get faster every 18 months without effort. Like all other programmers, simulator developers will need to discover how to get more performance from more cores. Second, the design space of possible future multicore architectures is much larger, so it's much harder to develop a standard simulator that many can use to do architecture investigation and multiprocessor architecture research is a niche activity that is unlikely to be supported by a commercial simulator. Third, parallel application performance can be very non-deterministic and multiple different runs of the same code might be required to gain confidence in reported performance numbers [7]. Fourth, to cope with power limits, microprocessors now use many techniques to improve power-performance that will require extended simulation time. For example, dynamic voltage scaling and frequency scaling per core means that clock cycles and instructions per clock cycle are no longer accurate measures of performance. Recent Intel microprocessors even offer "Turbo" modes that will allow some cores to run at much higher clock rates temporarily until temperature limits are reached. These modes will be turned on opportunistically by the hardware, without even the operating system being aware. The time constants for studying power and temperature will

require very long simulations: from 1 to 100 seconds of target time.

Given the multicore revolution, we claim that the biggest architectural research challenges now deal with multiple processors rather than increasingly sophisticated single processors. Indeed, there is even a commercial movement towards “manycore” architectures which use many simpler processors, such as the IBM Cell, Intel Larrabee, and Sun Niagara [8, 9, 10]. Hence, architecture investigation now needs to be able to look at many processors in addition to memory hierarchy and processor design. The number of cores per chip, sophistication of these cores, and even the instruction sets of the cores are all open to debate. Issues that have received little recent attention, like on-chip interconnect, are vital. Moreover, power is at least as important a resource to conserve today as chip area was in the past.

Although many architects likely agree with the first few paragraphs of this section, they may not agree that the old simulation software assumptions no longer hold. Software researchers and practitioners are trying to address the grand challenge of the multicore revolution: to make it easy to write programs that are efficient, portable, correct, and scale as the number of cores per microprocessor increases biennially as it has been to write programs for sequential computers [1]. Hence, old programs and operating systems are being rewritten to be compatible with increasingly parallel microprocessors. New programming models, new programming languages, and new applications are being invented, and given the urgency of the multicore challenge, architects cannot ignore them. Creating portable parallel programs that maintain high-performance has long been a challenge, so techniques like autotuning are becoming popular [11]. Rather than thinking of the program as a static object, autotuning adapts the program to the features of computer on which it is running and perhaps even to the input data to the program. Such self-adapting can happen at the time the computer is announced, at the time software is installed onto a particular example of that computer—depending on the number of cores, clock rate, amount of memory, and types of compilers installed—or even during the execution of the program.

Software Architecture Model Execution: SAME

The performance challenge for software simulators is turning the increasing number of

host cores into higher simulated target instructions per second. We believe the challenge will be far harder for detailed simulation than for functional simulation, as there is naturally much more communication between components in a target cycle.

The opportunity for a “Software Architecture Model Execution” (SAME) simulator is to leverage the natural parallelism in the target machines to run the simulator faster on the multicore hardware of the host machine. The difficulty of this challenge is demonstrated by the Electronic CAD industry. Despite the availability of parallel servers for more than a decade, and even though there is massive potential parallelism in the system being simulated, most of the state-of-the-art register-transfer language (RTL) simulators are still not parallelized. Note that the market for RTL simulators is likely much larger than for architecture simulators, so it’s hard to be hopeful for commercial progress on this topic.

An alternative approach would be to use a cluster to emulate a multicore computer, perhaps with one node of cluster for each processor, and then use the cluster network for communication between cores. As most clusters typically interconnect via hierarchies of Ethernet switches, a major challenge will be to prevent the network from becoming a bottleneck. We believe the issue is more the latency of synchronization rather than data bandwidth. For example, the Wisconsin Wind tunnel project spent 40% of the time in the network in their simulations [12]. Since they were using the low-latency Myrinet network and much slower computers, we expect it to be much worse today given higher latency Ethernet networks and multi-gigahertz processors. Once again, we expect the challenge will be greater as the level of detail required increases. While we encourage others to make progress on this important but difficult problem, we are more excited by an alternative approach. As observed by the Research Accelerator for Multiple Processors (RAMP) Project [16], FPGA technology offers a promising vehicle for architectural investigation.

FPGA Architecture Model Execution Simulators: FAME

The first advantage is that a program written in a Hardware Description Languages (HDL) like Verilog or VHDL naturally runs in parallel on an FPGA, while a program written in languages like C or Java naturally runs sequentially on a computer. Indeed, the programmer needs to do extra work to prevent parallelism in an HDL on an FPGA. A second

advantage is relative to hardware development, changing an FPGA is very fast. You only need go through the sequence of CAD programs and download the contents into a FPGA, which can take less than one hour for a small design. Hence, you can “tape out” a new design every day, as opposed to every three to six months with real hardware. (You also don’t have to pay the millions of dollars for new masks and to fabricate chips.) This flexibility also makes it easy to add runtime measurements without slowing down simulation since FPGAs are naturally parallel. Third, the FPGA hardware is relatively inexpensive. At the low end is a \$750 board with a small FPGA (Vertex 5 XUP board) to a \$15,000 board with four medium-sized FPGAs (BEE3). Fourth, relative to software simulators, FPGAs are fast: today’s FPGA-based simulators run at about 100 target MIPS with timing models. Finally, Moore’s Law applies to FPGAs as well as to microprocessors, so FPGAs are roughly doubling in capacity every 2 years. Hence, rather than a processor taking many FPGAs as in the past, depending on processor complexity and size of FPGA, now many processors can fit on a single FPGA. Looking forward, we would expect the number of cores per FPGA to double roughly every two years. This trend is a great match to the multicore challenge.

There are downsides to using FPGAs as well. Simulator developers need to learn a HDL and FPGA CAD tools. Sometimes complex verification techniques are required to successfully complete an FPGA design, such as an equivalence checker. Moreover, FPGA CAD tools are of poorer quality than software tools (or even ASIC tools), so they can be frustrating to use. While such problems led universities to make their own CAD tools in the past, the FPGA industry keeps internal formats proprietary, thereby preventing third parties from developing their own CAD tools. Finally, compared to hardware, logic is inefficient in FPGAs unless you can find equivalent hardware primitives on FPGAs, such as DSPs and RAMs. In practice, FPGAs are bad at logic such as pipeline forwarding but good at state storage such as register files and caches.

Description of RAMP Gold

An example of Multithreaded FAME is RAMP Gold [13]. RAMP Gold v1.0 is implemented on a single Xilinx Virtex5 LX110T FPGA. The FPGA board connects to a PC server through a 1 Gbps Ethernet link. This front-end server is responsible for all simulation controls, such as loading executable binaries and dumping simulation statistics. The front-

end machine also serves complex system calls whose functionality is not implemented in the simulated target software kernel, such as file I/O.

Inside the FPGA, we currently use a single-channel 233 MHz DDR2 memory controller based on the BEE3 memory controller [14]. It supports up to a 2GB dual-rank SODIMM. We use the 2GB DRAM for both target memory and some simulated microarchitectural state, such as target cache tags and data. On top of the memory controller, there is a host cache whose purpose is only to accelerate the simulation—it does not affect the timing of the target memory system. The simulation engine includes two basic models: a processor model and memory system model. Each processor model emulates a single in-order issue 32-bit SPARC V8 CPU. The functional model is built on our previous work in [15], which is highly optimized for the Xilinx Virtex 5 family of FPGAs and runs at over 100 MHz. Every functional model simulates up to 64 target processors using host multithreading. The functional model implements the full SPARC V8 ISA, including floating-point and precise exceptions. It also has been verified against the SPARC V8 verification suite. All integer instructions, double precision floating-point multiply, add/subtract and conversions are implemented in hardware. Complex floating-point operations, such as FDIV, cause traps and are emulated in the simulated supervisor. The timing model emulates a classic five stage pipeline. For instance, it models pipeline stalls such as the load-use delay and the branch delay slot. The number of cores being simulated can be configured at runtime without resynthesizing the simulator.

The processor-timing model is connected to a configurable model of the memory hierarchy. The target system has split first-level instruction and data caches connected to a unified L2 cache, which can be configured as private or shared. Many cache parameters, including cache size, line size, associativity, and hit latency, are configured at runtime; within reasonable limits, varying them does not require resynthesis. In the current version of the model, the write-back and write-allocate policies are fixed, and replacement is pseudorandom; these restrictions are only for design simplicity and are not the result of inherent limitations in RAMP Gold. In RAMP Gold v1, the target caches are kept magically coherent because the underlying host is coherent; protocol transitions are not modeled, nor is contention for the interconnect. Constructing a cycle-accurate model of a coherence protocol is among our planned future work. The underlying memory functional model keeps our shared memory design

functionally correct, even if timing is incorrect. As with software simulators, timing model validation will be required to ensure reasonable accuracy is achieved. In addition to hardware simulation models, RAMP Gold also provides a systematic design and verification environment. Our target compiler tool chain is directly built from the latest GCC without any modification. Newlib provides a lightweight C library that is ABI compatible with OpenSolaris, so the same single-threaded application binary can run on RAMP Gold and Sun servers. Multi-threaded binaries are object-code compatible but must be linked against a different implementation of POSIX threads.

An Analysis of Ramp Gold

Blackscholes and Swaptions are two Intel RMS benchmarks from the Princeton Application Repository for Shared-Memory Computers (PARSEC) benchmark suite used to model speedup and workload of Ramp Gold. The Blackscholes application calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation (PDE). Swaptions uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions. Figure 1 shows the increase of speedup as the the number of cores increase of the Blackscholes application. Speedup is defined as the number of cycles one processor divided by cycles of n processors.

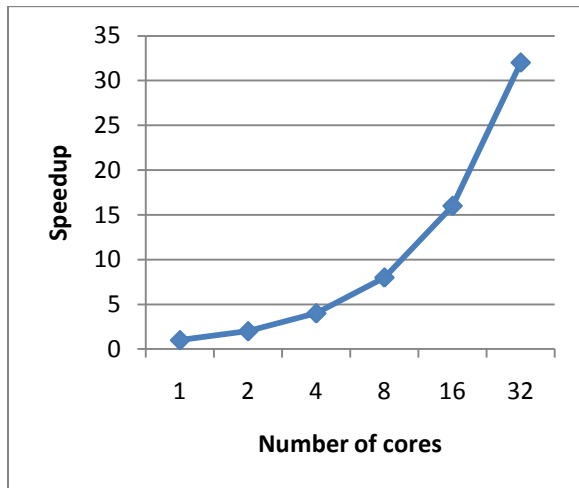


Figure 1 Speed up/number of cores of the Blackscholes benchmark program.

Figure 2 shows the impact on the working set as the cache size increases. The working set is the total number of cache misses (load misses plus store misses) divided by the total number of load and stores.

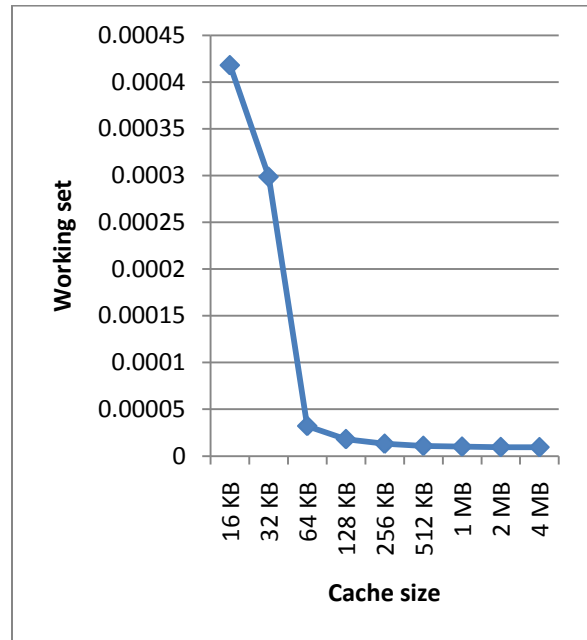


Figure 2 Working set/cache size of the Blackscholes benchmark program.

Figure 3 shows the speed up curve with respect to the number of cores for the Swaptions benchmark.

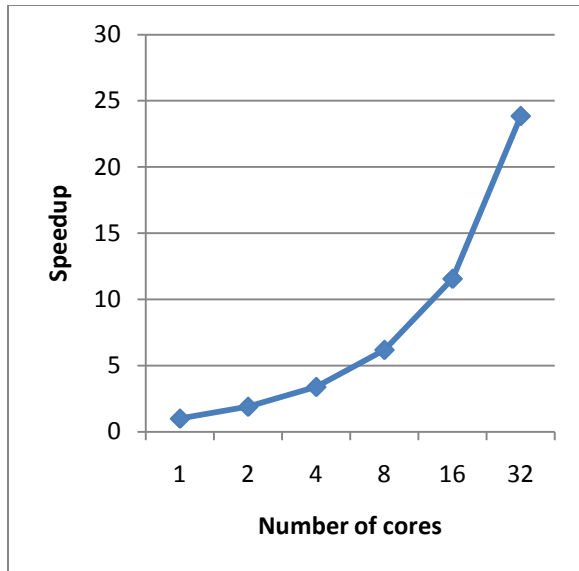


Figure 3 Speed up/number of cores of the Swaptions benchmark program.

Conclusion

These preliminary results show trends that were expected. Speedup increases when you increase the number of cores in a multicore system. This is expected in a multicore system with a constant workload. With more cores available more instructions can be carried out in parallel. This means an increase in performance.

The Working set decreases in plateaus as the cache size increases. When dealing with memory, a greater cache size means a smaller region for error in the form of cache misses. With few cache misses there are less main memory calls which increases performance in a single processor.

These behaviors are seen in the Blackscholes and the speedup trend is seen the Swaptions. Further testing is necessary for the remaining programs in the PARSEC benchmarking suite to complete the data set to compare against SAME results.

Acknowledgements

I would like to thank the following people who made this paper possible: Andrew Waterman, Scott Beamer, Sarah Lynn Bird, Henry Cook, and Dr. Krste Asanovic

The Par Lab is supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227), and by donations from Samsung, NEC, National Instruments, Nokia, and NVIDIA. Krste Asanovic, Zhangxi Tan, Andrew Waterman, and David Patterson

References

1. K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams and K. Yelick, The Landscape of Parallel Computing Research: A View from Berkeley, Technical Report No. UCB/EECS-2006-183, December 18, 2006
2. D. Patterson, G. Gibson, R. Katz: A Case for Redundant Arrays of Inexpensive Disks (RAID). SIGMOD Conference 1988: 109-116
3. K. Asanovic, Z. Tan, A. Waterman, and D. Patterson, "A Case for FPGA Architectural Simulation Execution (FAME)", Unpublished draft manuscript, 2009.
4. D. Patterson, C. Séquin, "RISC I: A Reduced Instruction Set VLSI Computer," Proceedings of the 8th Annual International Symposium on Computer Architecture, Minneapolis, Minnesota, 1981. p. 443 – 457.
5. J. Hennessy, N. Jouppi, J. Gill, F. Baskett, A. Strong, T. Gross, C. Rowen, J. Leonard. The MIPS Machine. Twenty-Fourth IEEE Computer Society International Conference, San Francisco, California, USA, February 22-25, 1982
6. L. Daniel, J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. at the 17th International Symposium on Computer Architecture (ISCA-17): 148-159, Seattle, WA, June 1990.
7. A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiawicz, Beng-Hong Lim, Kenneth Mackenzie, Donald Yeung: The MIT Alewife Machine:

- Architecture and Performance. at the 22th International Symposium on Computer Architecture (ISCA-22): 2-13, Santa Margherita Ligure, Italy June 1995
7. A. Alameldeen, D. Wood, Addressing Workload Variability in Architectural Simulations, IEEE Micro Special Issue: Micro's Top Picks from Microarchitecture Conferences, November- December 2003
 8. M. Gschwind, H. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, T. Yamazaki: Synergistic Processing in Cell's Multicore Architecture. IEEE Micro 26(2): 10-24 (2006)
 9. L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, P. Dubey, S. Junkins, A. Lake, R. Cavin, R. Espasa, E. Grochowski, T. Juan, M. Abrash, J. Sugerma, P. Hanrahan, Larrabee: A Many-Core x86 Architecture for Visual Computing, IEEE Micro, January/February 2009 (vol. 29 no. 1) pp. 10-21
 10. P. Kongetira, K. Aingaran, K. Olukotun: Niagara: A 32-Way Multithreaded Sparc Processor. IEEE Micro 25(2): 21-29 (2005)
 11. E. Im, K. Yelick, R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," International Journal of High Performance Computing Applications, vol. 18, no. 1, Spr. 2004, pp. 135-158.
 12. S. Mukherjee, S. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. Hill, J. Larus, D. Wood, Fast and Portable Parallel Architecture Simulators: Wisconsin Wind Tunnel II, IEEE Concurrency, October-December 2000
 13. Z. Tan, A. Waterman, R. Avizienis, Y. Lee, D. Patterson, K. Asanović, "RAMP Gold: An FPGA-based Architecture Simulator for Multiprocessors," 4th Workshop on Architectural Research Prototyping (WARP-2009), at 36th International Symposium on Computer Architecture (ISCA-36), Austin TX, June 2009
 14. DDR2 DRAM Controller for BEE3, <http://research.microsoft.com/enus/projects/BEE3/>
 15. Z. Tan, K. Asanovic, D. Patterson, "An FPGA Host-Multithreaded Functional Model for SPARC v8", 3rd Workshop on Architectural Research Prototyping (WARP-2008), at 35th International Symposium on Computer Architecture (ISCA-35), Beijing, China, June 2008.