RAMP Gold: A High-Throughput FPGA-Based Manycore Simulator

Rimas Avizienis, Yunsup Lee, Andrew Waterman

Parallel Computing Laboratory, Computer Science Division
University of California, Berkeley
{*rimas,yunsup,waterman*}*@cs.berkeley.edu*

Abstract

Computer architects have long used simulators to explore microarchitectures and quantitatively analyze design tradeoffs. Though attractive because of their low cost and ease of modification, these software simulators suffer from poor performance. In this paper, we present RAMP Gold, an FPGA-based manycore simulator that outperforms software simulators by orders of magnitude. It models up to 64 in-order issue SPARC V8 cores and a shared memory hierarchy. We describe the implementation of RAMP Gold in detail and analyze its performance. We then conclude, proposing RAMP Gold's use in the exploration of several important research problems that currently suffer greatly from the poor performance of software simulators.

1 Introduction

Computer architects have traditionally evaluated instruction sets and microarchitectures by executing representative programs and measuring one or more figures of merit: clock cycles per instruction, execution time, and power are among the most common. Decades ago, researchers commonly prototyped their designs by fabricating integrated circuits and running benchmarks directly on the hardware. Albeit time-consuming, building a chip had several benefits: architects could run long-running, realistic programs on the hardware, and furnishing real implementations lent more credence to the value of their innovations.

As feature size shrank, transistor count soared, and clock rate rapidly rose, the VLSI design effort became intractible for most research groups. Many architects turned to software-based microarchitecture simulators to evaluate their designs. Although this shift reduced prototyping time dramatically, it came with the cost of a 10,000x performance penalty.

Contemporaneous with the sea change towards simulation was the advent of standardized benchmark suites. For a fixed compiler and instruction set, the benchmark binaries stayed the same. This fact allowed architects to employ a variety of techniques to cope with high simulation latency: multi-mode simulation, statistical sampling [16], and trace-based simulation, to name a few. (citations) Short, representative phases of benchmark programs, known as sim points, could be pre-determined, so portions of a benchmark could be simulated on a detailed model of the microarchitecture, while the lion's share of the dynamic instructions were only functionally simulated, or perhaps used to warm the caches.

Though expedient, treating benchmarks as static objects has the disadvantage of using yesterday's programs to evaluate tomorrow's architectures. Conversely, using tomorrow's programs eliminates the benefits of memoizing representative benchmark phases. This problem is exacerbated by the increasing popularity of autotuning [14], which changes program behavior–even algorithm selection–based upon subtle changes in the microarchitecture. The net result is that it will be difficult to capture meaningful performance data from new applications without simulating entire programs (and, hence, long instruction traces). Going forward, it will likely become intractable to explore ever-expanding design spaces with software simulators.

An additional challenge that software-based simulation has yet to overcome is the end of the era of ever-increasing single-thread performance [1]. Like other programs, software simulators doubled in performance every 18 months without any programming effort. In the multicore era, these performance gains must instead come from leveraging thread-level parallelism, something that microarchitecture simulators have not done with great success. Instead, researchers typically simulate multiple threads of execution with a single simulator thread, making simulations take even longer.

One alarming consequence of the high latency of software simulation can be observed by comparing papers accepted by ISCA in 1998 with those in 2008. In a ten-year period, the number of instructions simulated in a typical run increased only twofold. In this period, however, the chip multiprocessor became ubiquitous. Taking into account the number of simulated cores, the number

of instructions per core actually fell by almost a factor of two. Combined with a sixfold increase in processor clock frequencies in the last decade, the amount of simulated time has fallen by a factor of 11 in the last 10 years, from 400 ms to under 40 ms. Put another way, architects are now attempting to model the complex interaction of multiple processor cores by simulating programs for fewer than four OS scheduling quanta!

To help overcome this crisis in microarchitecture research evaluation, we present RAMP Gold, an FPGA-based architecture simulator that can track detailed microarchitectural state and timing information while executing at nearly 100 MIPS. RAMP Gold's significant speedup over software simulators is not only of utility to architects: it is only 100x slower than realtime, so we expect it to be additionally viable as a software development platform.

The rest of this paper is organized as follows. In Section 2, we discuss related software- and FPGA- based simulation techniques. In Section 3, we present the RAMP Gold simulator architecture. In Section 4, we describe its hardware implementation in detail. In Section 5, we discuss the infrastructure support we provide to make RAMP Gold a robust development platform. In Section 6, we evaluate the simulator's performance. In Section 7, we discuss our future directions and conclude.

2 Related Work

As architecture research began to focus more on implementation and less on instruction set design, it became unnecessary for every research group to maintain its own simulator. Undoubtedly, the popularity of software-based microarchitectural simulators is part and parcel of their standardization. Among the most widely used are SimpleScalar [2], ASIM [5], SESC [9], and RSIM [10]; additionally, there have been commercial efforts, like Simics[1] [7].

Field programmable gate arrays (FPGAs) have been employed at various levels of abstraction in the design, implementation, and evaluation of computer architectures. Several commercial efforts have attacked the slow speed of register transfer level (RTL) simulation by providing FPGA-accelerated logic simulation and verification. Quickturn was a notable early product in this vein; Cadence Palladium [3] is a modern example. These functional verification devices operate at relatively low speed–about 1 MHz–but should be contrasted with software-based RTL simulation, which may run at 1 KHz or slower on a fast workstation.

Rather than using FPGAs to simulate RTL, other projects have mapped processors directly to FPGAs, enabling higher clock rates. Early projects under the umbrella of the Research Accelerator for Multiple Processors (RAMP) fall into this category, namely RAMP Blue [6], a 1008-core message-passing machine, and RAMP Red [13], another multicore simulator used to investigate issues in transactional memory. Both projects used Xilinx FPGAs on Berkeley Emulation Engine 2 (BEE2) boards [4].

One disadvantage of direct RTL mapping is that timing is a property of the functional implementation. Some designers have augmented direct RTL-mapped systems with simple timing models by clock-gating various components to get correct behavior while faking the desired timing. RAMP Blue 2 [6] is an example of such a system. Unfortunately, such a system implies a tight coupling between the timing and function because the timing model is a wrapper around each functional component.

The lack of modularity in these direct RTL-mapped systems with timing wrappers led to FPGA implementations of simulators that separately model the function and timing of the target system, a trick that has been used in software simulators. HASIM [12], a hardware-accelerated port of ASIM, employs split timing and functional models.

3 RAMP Gold Architecture

Conventional wisdom in computer architecture led to the proliferation of similar uniprocessor designs, but there is not yet any conventional wisdom for manycore designs [15]. Our target machine model–a 64-core, single-socket, tiled CMP–is only one point in this vast design space, but it is a simple baseline microarchitecture that can be modified to explore a broad range of manycore topics.

A target core is a single-issue, in-order SPARC V8 with private L1 data and instruction caches. Up to 64 such cores are connected to a shared, unified L2 cache and coherence network. The L2 is pumped by one or more on-die memory controllers.

RAMP Gold, the host machine, models this target system with high performance and low cost by employing two important techniques. First, the timing and function of the target system are modeled separately. This design enables an arbitrary implementation of the timing and functional models as long as their interfaces are consistent, leading to the second technique, host-multithreading. We describe both in detail in the following sections.

[1] It is interesting to note, however, that SIMICS has deprecated support for instrumenting individual loads and stores, thus making it nearly useless for modeling the timing of the memory hierarchy. Simics is used mostly by OS and device driver writers; computer architects are merely a secondary market for this tool.

3.1 Split Functional and Timing Models

Like many architecture simulators, RAMP Gold splits its target model into two components: a functional model and a timing model. The functional model implements the target processor ISA and shared memory system, while the timing model controls when instructions are executed by the functional model and determines how many target cycles a given operation takes. Decoupling the functional and timing behavior of the target model in such a way allows the design to be modularized, and makes it possible to vary the timing dependent behavior of the target model without requiring any changes to the functional model.

Cache modeling provides one demonstration of the benefits of using split functional and timing models. In a target model without such a functional/timing split, the straightforward way to model a cache of a given size would be to directly implement it. The limited on-chip memory resources available in an FPGA would severely limit the maximum size of the cache that could be modeled without using off chip memory, and using off chip memory to model the cache would result in significantly reduced performance. With split timing and functional models, cache behavior can be simulated using a timing model which only maintains the cache state necessary to determine whether a given memory access would result in a cache hit or miss. Instead of tracking all the data resident in the modeled cache, the timing model only needs to store and update the cache tags and other metadata associated with each cache line. The functional model handles the execution of the load or store instruction, and the timing model causes the thread which performed the memory access to stall if it determines that the memory access would have resulted in a cache miss. This allows for the efficient simulation of much larger caches than would otherwise be possible. We have implemented such a cache model, whose parameters (within certain bounds) are set at run time. This enables us to simulate the impact of different cache organizations on program performance without requiring a resynthesis of the design.

Modeling a functional unit (for example, an FPU) with an arbitrary execution latency provides another example of the utility of the split functional/timing model paradigm. To model an FPU with a single target cycle latency in an FPGA based simulator without a functional/timing split would require an FPU which could execute a given operation in a single host cycle. This would necessitate the use of a low clock frequency for the entire pipeline, resulting in poor performance. However, using a split functional/timing model allows the execution of a single target cycle to be emulated in any number of host cycles. This permits the use of a pipelined FPU with a multicycle latency, and therefore doesn't require a dramatic reduction of the clock frequency of the pipeline.

3.2 Host Multithreading

Previous FPGA based multiprocessor simulators have mapped many instances of processor designs implemented in RTL directly to FPGAs. For example, the RAMP Blue project mapped 1008 32-bit Xilinx MicroBlaze RISC cores onto a rack of 21 BEE2 boards (each consisting of 4 Xilinx VirtexII Pro 70 FPGAs, resulting in 12 cores per FPGA) to emulate a manycore distributed-memory message-passing target architecture [6]. RAMP Gold takes a different approach. Instead of implementing many physical copies of a target processor, it time-multiplexes multiple hardware thread contexts onto a single target model–a technique we call *host multithreading*. We emphasize that a host-multithreaded simulator need not model a multithreaded target.

Using host multithreading to share a single execution engine among many virtualized processors has many benefits for the RAMP Gold design. It enables an area efficient FPGA implementation, as a single pipeline can be used to model the behavior of many target processors. Our current implementation simulates up to 64 processor cores using a single physical pipeline, and multiple such pipelines can fit on a single FPGA. Additionally, host multithreading allows a deeply pipelined execution engine to be used without the need for any bypass paths. Since at most one instruction from each virtual processor is in the pipeline at any point in time, there is never a need to transmit a result from one pipeline stage to an earlier stage. The lack of bypass networks and forwarding logic in the critical path means the pipeline can run at a higher clock frequency than would otherwise be possible. Host multithreading also effectively hides many of the latencies encountered during a simulation. For example, handling a cache miss doesn't require stalling the entire pipeline. The other hardware thread contexts can continue executing and by the next time the context which caused the miss is scheduled to execute, the cache miss will likely have been resolved. The net result is high simulation throughput.

4 RAMP Gold Microarchitecture

In this section, we describe the implementation of RAMP Gold's functional pipeline, memory system, floating point unit, pipeline inspector interface and front end link.

4.1 Pipeline

Figure 1 illustrates the architecture of RAMP Gold's functional pipeline. The pipeline is deep to achieve low clock period; provided that the target machine has at least as many threads as there are stages, it is hazard-free. The pipeline supports up to 64 independent SPARC V8 hardware contexts through the use of fine-grained hardware

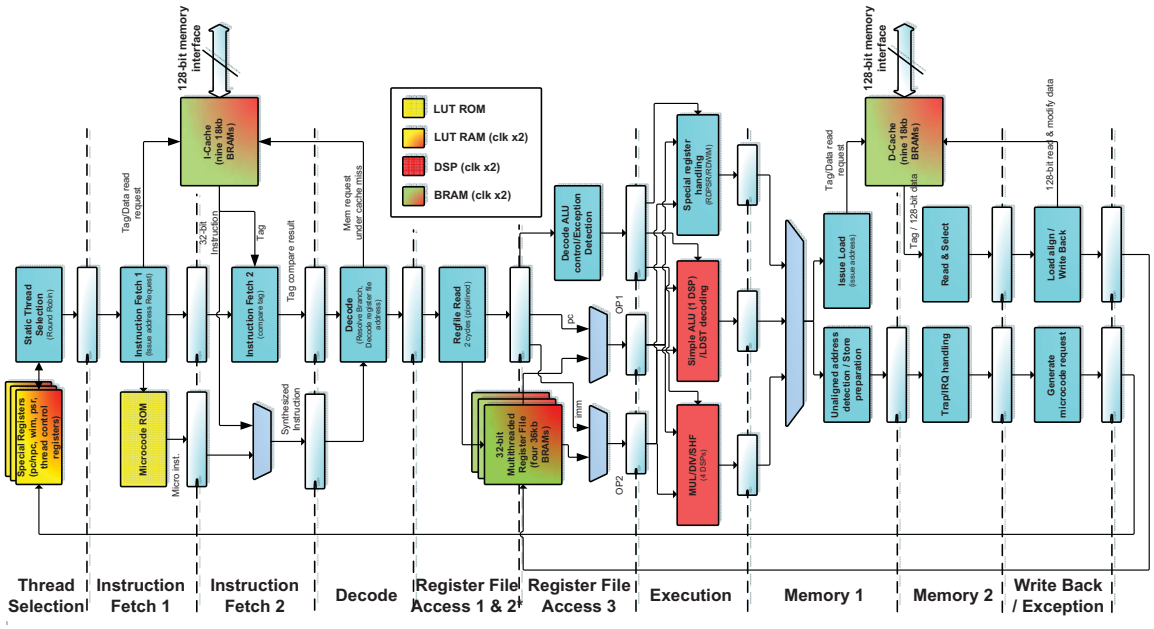

Figure 1: RAMP Gold Integer Pipeline

multithreading. Instructions from different threads are issued by a static round-robin scheduler. When a long latency instruction (such as a host cache miss) is encountered, the instruction is not committed but is instead reissued on that thread's next fetch turn. If this 'replayed' instruction has completed by its second trip through the pipeline, it is then able to commit. Ideally, long-latency events can be sufficiently hidden by the interleaved execution of other threads that the commit rate is kept high.

Internally, the functional pipeline only directly implements a subset of the SPARC V8 ISA. Microcode is used to handle complex instructions that require a sequence of operations (e.g. atomic operations such as SWAP) and infrequent operations such as traps. Although using microcode adds some complexity to the decode logic, it eliminates the need for more expensive structures in the pipeline and simplifies the overall design. For example, direct implementation of the SPARC indexed-store instruction would require a register file with three read ports, which, on the Xilinx Virtex 5 FPGA, doubles its size. Instead, we keep the register file small by implementing indexed stores using two microcode instructions: an effective address calculation and a store.

Our register file implementation supports up to 7 register windows. The contents of the register file for a single thread occupy 128 32-bit words in block RAM. Eight of these words are used to store ancillary processor state and microcode-mode registers. The 64-way multithreaded register file consumes eight 36Kb block RAMs. Other special architectural registers such as the PC, nPC and processor state register are mapped to distributed LUTRAMs.

To accelerate digital signal processing, most FPGAs include some "hard" DSP blocks, which have become progressively more feature-rich over successive FPGA generations. In the Virtex 5, each DSP block is capable of performing 48-bit two's complement addition and subtraction as well as bitwise logic operations and 48-bit pattern detection. We have found that the computation required by most SPARC instructions (including all simple ALU operations and effective address calculations) can be performed using a single DSP block. The pattern detector is used to generate the integer condition codes.

Currently, the functional pipeline consumes 14% of the LUT resources and 23% of the block RAM resources on a single Xilinx Virtex5 LX110T FPGA. We believe that we can fit three pipelines in one FPGA, allowing us to simulate up to 192 target cores.

4.2 Memory System

We currently use a single-channel DDR2 memory controller running at 233 MHz which is based on the BEE3 memory controller design [8]. It supports a dual-rank SODIMM up to 2GB in size. At present, we use the 2GB DRAM only for target memory, though in the future we envision that microarchitectural state such as target cache tags might also be stored in DRAM.

To reduce memory traffic and improve performance, we built a small per-thread host instruction cache and a unified host data cache. For the sake of simplicity, the

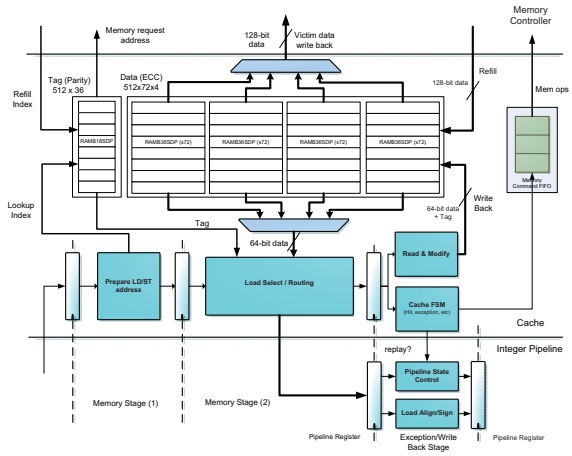

Figure 2: RAMP Gold Host Cache

data cache is direct-mapped and uses a write-back/write-allocate policy. The cache line size is 32 bytes to match the minimum DDR2 burst size. Each of the 64 host threads is allotted a 256-byte instruction cache. The unified data cache is 16KB in size and is shared between all threads. As a result, data in the cache is automatically kept coherent. The data cache supports multiple hits under a single miss. In order to support the SWAP instruction (which is implemented using three microcode instructions that must be executed atomically), the data cache locks the cache line until the SWAP instruction has retired. The integration of the data cache with the pipeline is illustrated in Figure 2. The I & D caches (including tags) are mapped to two 18Kb block RAMs and eight 36Kb block RAMs.

4.3 Floating-Point Unit

In order to accelerate floating point operations, we integrated a floating-point register file and functional units which perform single- and double-precision add and multiply operations into the pipeline. Load, store, add, multiply, and other simple floating-point instructions are handled entirely in hardware; complex instructions, namely integer conversion, divide, and square root, generate a trap and are emulated in supervisor mode.

Our current design uses Xilinx Floating Point IP blocks to implement the add and multiply functional units. These blocks map efficiently onto the DSP units available on the FPGA, but unfortunately are not 100% IEEE 754 compliant; consequently, we trap on denormalized numbers and emulate the offending operation in the supervisor.

The floating point functional units extended the pipeline depth to 16 stages.

4.4 Pipeline Inspector

The pipeline inspector is an interface used by the front-end machine to inject instructions directly into the pipeline and read back their results without affecting target timing. This powerful feature allows the front-end machine to read and write the target memory and processor state, which facilitates debugging software running on the target. In the future, we plan to implement simulation checkpointing using this functionality. The pipeline inspector can also be used to retrieve target model performance information. Each target model includes a number of 64-bit performance counters which log events such as the number of retired instructions. These counter values can be read by an application running on the target as well as the front end machine.

4.5 Front-end Link

The FPGA board connects to a front-end machine through a gigabit Ethernet link. This link provides a communication channel between the front-end machine and the pipeline inspector, giving the front end machine full control over the simulator and the target model.

5 Infrastructure

The RAMP Gold infrastructure includes a number of development and debugging tools which combine to form a complete and versatile design and verification environment. A recent version of the SPARC GCC toolchain and a runtime environment which uses a proxy kernel to emulate basic OS functionality enable RAMP Gold's users to build and run C and C++ programs without any modifications. Using this infrastructure we were able to build and run the SPEC CPU2000 and SPLASH2 benchmarks, as well as Damascene (part of the ParLab content-based image retrieval application) on RAMP Gold. Figure 3 summarizes our infrastructure.

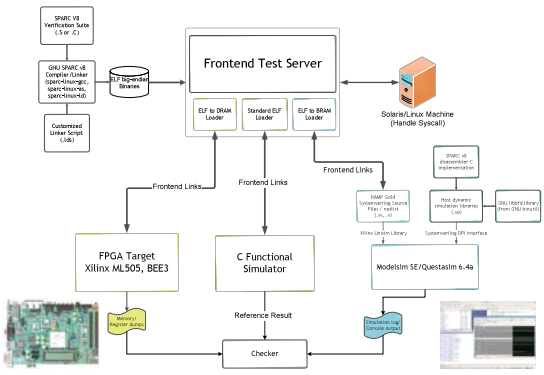

Figure 3: RAMP Gold Infrastructure

5.1 Toolchain

Our toolchain is built directly from the latest version of GCC without any modifications. GCC is compiled to use our modified version of newlib (a C library intended for use on embedded systems). We have extended newlib to support multithreaded applications and run C++ code. We wrote our own implementations of the 17 system calls required by newlib, recursive locks to support thread-safe C library functions, and execution startup routines (crt0.s) for C++ programs. Single-threaded binaries compiled with our toolchain are ABI-compatible with OpenSolaris, and as a result the same 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 to run on Sun machines. The RAMP Gold development environment provides bare metal access to hardware threads via the HardThreads API [11]. Synchronization primitives such as barriers and locks are provided and the BLAS and LAPACK libraries have been ported to support scientific computing applications.

5.2 Proxy Kernel

The proxy kernel handles bootstrapping, traps and system calls. The bootstrapping code initializes stack frames for each thread, copies command line arguments to thread 0's stack, and transfers control to it. Thread 0 starts running user code, while the other threads spin until they receive an active message. Active messages are currently implemented by polling special memory locations, though we intend to implement them using interrupts in the near future. When an active thread wants to spawn a new hardware thread, it packs the thread id and target PC into an active message and writes it to a designated location in memory. When an active message is delivered, the receiving thread jumps to the target PC address. When the thread is done executing, it returns to the polling loop.

Whenever the user code traps due to an illegal action or a system call, the PC is set to the trap handler which resides in the proxy kernel. The floating point emulation code also lives in the proxy kernel and is invoked when a floating point trap occurs. If a system call cannot be serviced locally, the proxy kernel issues a request to the front-end machine to handle it and waits for the result.

5.3 Front-End Machine

The front-end machine uses the pipeline inspector interface to load a program and command line arguments into the target's memory and signal it to begin execution. It periodically polls a target memory location to determine whether the proxy kernel has requested that it handle a system call. Whenever the front-end machine receives such a request, it reads the system call arguments from the target memory, executes the system call locally, writes the results back to the target memory, and signals the target to proceed.

The front-end machine can communicate with three different implementations of the target architecture: C-Gold (our in-house functional simulator written in C), the SystemVerilog RTL code simulated using Modelsim, and the synthesized design running on the FPGA. When developing or porting a program, we verify that it is functionally correct by running it on C-Gold prior to trying it out on hardware. While we are developing RTL code, we iterate between simulating it on Modelsim and testing it on the actual hardware. These multiple implementations of the target architecture allow us to get debugging information at any level of detail we need.

5.4 Applications: Damascene

We successfully compiled and ran the C version of Damascene on RAMP Gold using our infrastructure. This application takes advantage of bulk synchronous parallelism (which is a natural fit to the hard threads model), uses floating point operations heavily, calls BLAS and LAPACK routines, and has a 1.2GB working set. We have stress tested the RAMP Gold hardware (including the pipeline and the memory subsystem) and the infrastructure by running this application for extended periods of time.

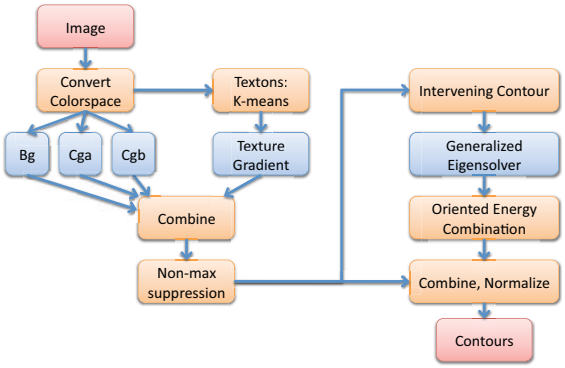

Figure 4: High level pipe and filter diagram of Damascene

Figure 4 outlines the high level pipe and filter diagram of the application. The first module takes the input image and produces four channels (luminance L, chrominance a, chrominance b, and texton). The second module of the algorithm extracts a set of local cues from each of the input channels. In the third module, these local cues are linearly combined into a single multiscale probability boundary signal (mPb). An affinity matrix W (which

encodes the similarity between pixels) is constructed by using the intervening contour cue which is the maximum value of mPb along a line connecting two pixels. The affinity matrix is then used to calculate a set of eigenvectors and eigenvalues which represent segmentations of the image. The contours embedded in the eigenvectors are extracted by using a bank of eight Gaussian directional derivative filters. The information from the different eigenvectors and eigenvalues is combined into a spectral probability boundary signal. Finally, the local cue information and spectral information are combined to form a global probability of boundaries.

6 Simulator Performance

The primary goal of RAMP Gold is to accelerate the rate at which architects can simulate manycore systems, and as such simulator performance is the key metric with which we evaluate its success. In this section, we examine the performance of existing software simulators and measure RAMP Gold's performance while running a microbenchmark and programs from the SPLASH2 benchmark suite.

6.1 Software Simulator Performance

Simics is a popular architecture simulator which has been often been used by architects to prototype and evaluate new microarchictectural ideas. As such, it provides a reasonable baseline against which to compare RAMP Gold's performance. Running on a high end workstation and simulating 16 SPARC processor cores in a purely functional mode, Simics achieves approximately 8 MIPS of simulated performance. In a comparable configuration (with the timing model disabled), RAMP Gold's performance is over an order of magnitude faster, clocking in at 85 MIPS. Modeling the timing behavior of a cache in Simics (using the gcache module) slows performance tenfold to 0.8 MIPS. This is akin to using RAMP with the timing model turned on, which runs at 45.6 MIPS. These results are summarized in Figure 5.

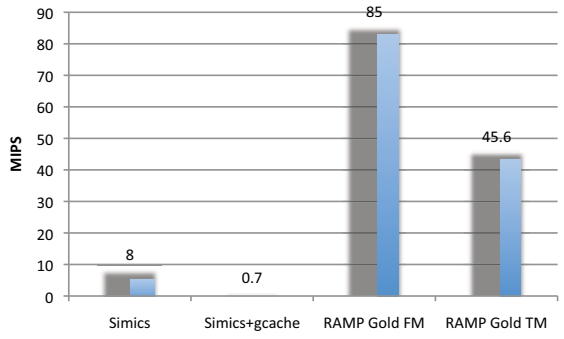

Figure 5: Simulator Performance comparison

6.2 RAMP Gold Performance

RAMP Gold's functional pipeline currently runs at 100 MHz. As the functional pipeline is single-issue, the upper bound on simulator performance is 100 MIPS. This level of performance can only be achieved, however, if there are no host stalls. Host cache misses and other long-latency simulator events, such as contention for the shared integer multiplier, reduce functional simulation performance. To evaluate RAMP Gold's performance in the absence of these performance-limiting factors, we wrote a synthetic, compute-bound benchmark which causes few host cache misses and does not use multi-cycle instructions.

Figure 6 shows the simulator's execution performance on this synthetic microbenchmark for power-of-two simulated core counts between 1 and 64. In functional-only mode, we achieve the full simulator throughput of 100 MIPS when the number of target cores can cover the functional pipeline depth of 16. For fewer target cores, the fraction of peak performance achieved is given by $\frac{\#\ cores}{pipeline\ depth}$.

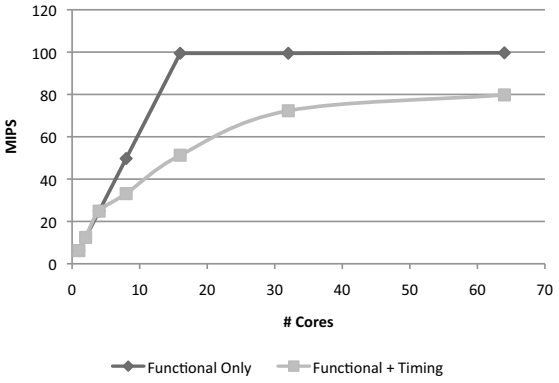

Figure 6: Synthetic CPU-bound Workload Performance

One function of the timing model is to keep all host threads lined up on the same target cycle. Since some instructions may replay through the functional pipeline (e.g. because of a host cache miss), the timing model does not allow the functional model to *issue* instructions corresponding to target cycle $n + 1$ until all instructions corresponding to target cycle n have *committed*. As a result, every simulated target cycle has one functional pipeline depth of wasted host cycles. This performance penalty is shown in Figure 6. The fraction of peak performance achieved is approximately $\frac{\#\ cores}{\#\ cores+pipeline\ depth}$. Thus, when simulating 64 cores, we achieve 80 MIPS on our synthetic benchmark, but only 50 MIPS with 16 cores.

Instructions that must replay through the functional pipeline limit simulator performance. Complex instructions like multiply, divide, and indexed-store require at

least one replay. Moreover, the host memory hierarchy can itself become a bottleneck, as host cache misses also cause replays. Unsurprisingly, we found that on realistic workloads, the unified 16KB data cache is inadequate when more than a single core is being simulated. Conflict misses are the primary reason for this: if two threads attempt to consume conflicting cache lines at the same time, then they will both miss once per load or store, rather than once per cache line.

We run a subset of the SPLASH2 benchmark suite with 16 cores and the timing model enabled to quantify RAMP Gold's performance on realistic workloads. FMM and Ocean are full applications (a fast multipole solver and an ocean simulator) while FFT and Cholesky are computational kernels (a complex 1D FFT and a blocked sparse Cholesky factorization). Figure 7 shows RAMP Gold's performance on these benchmarks for various target L1 data cache sizes. Since we are running 16 threads, peak functional performance (100 MIPS) would be about 6.7 million target cycles per second. As we are currently limited by the coupling of the timing and functional model described above, the peak performance with 16 threads is about 3.3 million target cycles per second. Cholesky nearly achieves this bound, but the other benchmarks suffer from varying degrees of conflict misses in the host cache.

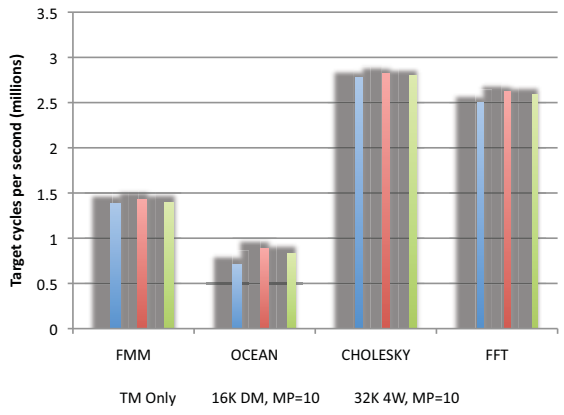

Figure 7: SPLASH2 Benchmark Performance

Figure 7 additionally echoes the benefits of split function and timing in two ways. First, the same synthesized design was used to model multiple target cache sizes. Second, the simulator's performance is approximately independent of target machine parameters.

We further investigate the performance impact of the unified host cache by running Ocean with a fixed problem size while varying the number of simulated cores. The performance results are plotted in Figure 8; the performance of the synthetic compute-bound workload are provided as well for comparison. Peak performance

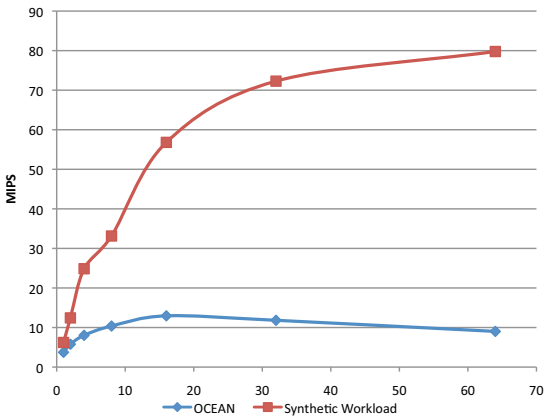

Figure 8: SPLASH2 Ocean Performance

occurs at 16 cores when the functional pipeline's latency can be covered. Increasing the number of cores beyond the pipeline depth makes the simulator performance worse because the cache miss rate is even higher[2]. Even at the peak at 16 cores, however, the synthetic workload realizes four times the throughput.

6.3 Improving Simulator Performance

RAMP Gold performs well when executing programs that do not destructively interfere in the host data cache. Sadly, data-parallel workloads tend to have access patterns that pathologically cause host cache conflicts. We plan to attack this problem in two ways: by reducing the miss penalty and reducing the miss rate.

Although the unified host cache can service a hit under a miss, it blocks on a second outstanding miss. Especially in the presence of a high host cache miss rate, this policy effectively serializes multiple threads' memory accesses, eliminating the latency-hiding benefits of host-multithreading. Building an MSHR file and servicing multiple misses in parallel will significantly reduce the host cache miss penalty.

We can reduce the host cache miss rate in one of two ways without increasing its on-chip memory footprint. On one hand, we could make it set-associative. Unfortunately, high associativity may be needed to significantly reduce the miss rate. On the other hand, we could give each thread a private host cache and keep them coherent. This option will likely increase the miss penalty because of the need to examine extra state on a miss, but it has the benefit of not requiring a nonblocking cache for reasonable performance.

The host cache design space is large. In the coming weeks, we plan to use RAMP Gold to simulate itself to

[2]Although the simulator's performance falls as the core count is increased, the number of target cycles required to execute the benchmark falls, as one would expect.

explore this space so that we can bridge the gap between realized and peak simulator performance.

7 Conclusions and Future Work

We conclude by proposing three exciting research opportunities that would be intractable with traditional software-based architecture simulators but are enabled by RAMP Gold.

As microarchitectures have become increasingly complex, autotuning has become a popular means by which to obtain high performance for a given kernel on a variety of hardware implementations. A compelling extension of autotuning is hardware-software cotuning. The problem, stated simply, is to find the tuple of microarchitecture and kernel variant that maximizes some metric (e.g. FLOPS) subject to some constraints. Unfortunately, the design space is extremely large, suffering from combinatorial explosion even with a limited number of architectural and algorithmic parameters. Techniques like statistical sampling that have been used to speed software simulations do not apply in this scenario because the binary changes with the microarchitecture. With RAMP Gold, we can explore dozens of trillion-instruction program runs in a single day on a single FPGA to cope with state explosion and make hardware-software cotuning a reality.

Modeling the thermal behavior of CPUs is an important but computationally expensive problem. Thermal time constants are on the order of tens of milliseconds, so to model complex thermal effects, several seconds must be simulated at a high level of microarchitectural detail. Much of the analysis can ordinarily be performed offline, but if we wish to tackle the problem of operating system policies for thermal management, then we need feedback from the thermal model to the simulated operating system. To run such simulations in a reasonable amount of time, we propose to perform a full-system simulation on RAMP Gold and run a thermal model on a GPU on the front-end machine. Every few thousand target cycles, a package of activity factors is exported to the GPU, and up-to-date thermal information is sent back to the simulated OS. Such a simulation will not impact RAMP Gold's high performance and will allow simulations on a thermal timescale to run in only tens of minutes.

In a manycore system, the DRAM is a key shared resource whose management is critical to system throughput. Current software simulators cannot tractably model instruction traces of sufficient length to allow architects to confidently draw conclusions about DRAM scheduling policies. To cope with this limitation, previous work has simulated only a small number of cores and often uses trace-based simulation with a cache model. Although trace-based simulation can be accurate for in-order issue cores, it cannot accurately model programs that synchronize through shared memory. As we can tractably simulate a large number of dynamic instructions without resorting to traces, this area of research should be revisited in greater detail with RAMP Gold.

In RAMP Gold we have built a high-performance microarchitecture simulator that eclipses the performance of software-based simulators by orders of magnitude. It is even fast enough to serve as a software development platform. We plan to utilize RAMP Gold in our manycore architecture research and consider the above three problems to be part of our exciting future work.

8 Acknowledgements

We'd like to thank Zhangxi Tan, David Patterson, and Krste Asanović for their invaluable help and guidance on this project. We also thank Chuck Thacker and Microsoft for the BEE3 DRAM controller, and Mentor Graphics and Synopsis for the quick turn-around on ECAD tool bug fixes. Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227).

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, 2002.

[3] Cadence Design Systems. Palladium Accelerator/Emulator. http://www.cadence.com/products/functional_ver/palladium/.

[4] C. Chang, J. Wawrzynek, and R. W. Brodersen. BEE2: A High-End Reconfigurable Computing System. *IEEE Design & Test of Computers*, 22(2):114–125, 2005.

[5] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A Performance Model Framework. *Computer*, 35(2):68–76, 2002.

[6] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P.-Y. Droz. RAMP Blue: A Message-Passing Manycore System in FPGAs. In *International Conference on Field Programmable Logic and Applications*, August 2007.

[7] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35, 2002.

[8] Microsoft Research. Berkeley Emulation Engine 3. http://research.microsoft.com/en-us/projects/BEE3/.

[9] P. M. Ortego and P. Sack. SESC: SuperESCalar Simulator. Dec 2004.

[10] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In *In Proceedings of the Third Workshop on Computer Architecture Education*, 1997.

[11] H. Pan, B. Hindman, and K. Asanovic. Lithe: Enabling Efficient Composition of Parallel Libraries. In *Workshop on Hot Topics in Parallelism (HotPar-09)*. USENIX, March 2009.

[12] M. Pellauer, J. Emer, and Arvind. HAsim: Implementing a Partitioned Performance Model on an FPGA, 2007. http://publications.csail.mit.edu/abstracts/abstracts07/pellauer-abstract/hasim.html.

[13] N. N. Sewook, S. Wee, J. Casper, J. Burdick, Y. Teslyar, C. Kozyrakis, and K. Olukotun. Building and Using the ATLAS Transactional Memory System. In *in Proceedings of the Workshop on Architecture Research using FPGA Platforms, held at HPCA12. 2006*, 2006.

[14] S. Williams. Autotuning Performance on Multicore Computers, PhD thesis. Technical report, U.C. Berkeley, 2008.

[15] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.

[16] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Statistical sampling of microarchitecture simulation. *ACM Transactions on Modeling and Computer Simulation*, 16(3):197 – 224, 2006.