

COTSon: Infrastructure for Full System Simulation

Eduardo Argollo Ayose Falcón Paolo Faraboschi
Matteo Monchiero Daniel Ortega

HP Labs — Exascale Computing Lab

{eduardo.argollo, ayose.falcon, paolo.faraboschi, matteo.monchiero, daniel.ortega}@hp.com

ABSTRACT

Simulation has historically been the primary technique used for evaluating the performance of new proposals in computer architecture. Speed and complexity considerations have traditionally limited its applicability to single-thread processors running application-level code. This is no longer sufficient to model modern multicore systems running the complex workloads of commercial interest today.

COTSon is a simulator framework jointly developed by HP Labs and AMD. The goal of COTSon is to provide fast and accurate evaluation of current and future computing systems, covering the full software stack and complete hardware models. It targets cluster-level systems composed of hundreds of commodity multicore nodes and their associated devices connected through a standard communication network. COTSon adopts a *functional-directed* philosophy, where fast functional emulators and timing models cooperate to improve the simulation accuracy at a speed sufficient to simulate the full stack of applications, middleware and OSs.

This paper describes the changes in simulation philosophy we embraced in COTSon to address these new challenges. We base functional emulation on established, fast and validated tools that support commodity OSs and complex multi-tier applications. Through a robust interface between the functional and timing domain, we can leverage other existing simulators for individual sub-components, such as disks or networks. We abandon the idea of “always-on” cycle-based simulation in favor of statistical sampling approaches that can trade accuracy for speed.

COTSon opens up a new dimension in the speed/accuracy space, allowing simulation of a cluster of nodes several orders of magnitude faster with a minimal accuracy loss.

Categories and Subject Descriptors

B.2.2 [Arithmetic and Logic Structures]: Performance Analysis and Design Aids—*Simulation, Verification, Worst-case analysis*; B.3.3 [Memory Structures]: Performance Analysis and Design Aids—*Simulation, Verification, Worst-case analysis*

General Terms

Measurement, Performance

Keywords

Full system simulation

1. INTRODUCTION

The ever-increasing complexity of computing systems has made simulators the primary choice in their design and analysis. A correct simulation methodology helps researchers, developers and system designers understand the impact of their design decisions.

The panel discussion of the 2004 Intl. Symp. on Performance Analysis of Systems and Software (captured in [30]) presents five important suggestions: 1) allow for multi-processor and multithreaded simulation of operating systems (OS) and applications, 2) improve sampling techniques, 3) use higher-speed alternatives to cycle-accurate simulation, 4) select benchmark suites that contain representative and non-redundant benchmarks, and 5) make the methodology more robust and statistically based to allow for independent validation and to facilitate comparability. In this paper we present COTSon, a simulation framework that we believe addresses all of these challenges.

Simulation can be decomposed into two complementary tasks. *Functional simulation* “emulates” the behavior of the target system, including the OS and common devices such as disks, video, or network interfaces. An emulator is normally only concerned with functional correctness, so the notion of time is imprecise and often just a representation of the wall-clock time of the host.

Functional simulators must be precise and have been historically used to verify correctness of systems and to do early software development before the hardware is available. Recently some emulators (such as SimOS [21] or QEMU [4]) became fast enough to approximate native execution. These have evolved into virtual machines and hypervisors which have been used to isolate, consolidate, encapsulate and also provide hardware independence for application developers.

Timing simulation is used to assess the performance of a system. It models the operation latency of devices simulated by the functional simulator and assures that events generated by these devices are simulated in a correct time ordering. Timing simulations are approximations to their real counterparts, and the concept of accuracy of a timing simulation is needed to measure the fidelity of these simulators with respect to existing systems. Absolute accuracy is not always strictly necessary and in many cases it is not even desired, due to its high engineering cost. In many situations, substituting absolute with relative accuracy between

different timing simulations is enough for users to discover trends for the proposed techniques.

If we look at the trajectory of general purpose multicore processors [6, 10, 18], we can see that the number of cores per die is expected to grow quadratically with each new generation. Some specialized parts are already appearing with up to hundreds of cores [1, 3]. This trend broadens the variability of architectural design choices, such as cache hierarchy, heterogeneity, and use of lightweight cores. It also adds enormous pressure to the simulation infrastructure.

A defining characteristic of simulators is the control relationship between their functional and timing components [14]. In *timing-directed* simulation (also called *execution-driven*), the timing model is responsible for driving the functional simulation. The execution-driven approach allows for higher simulation accuracy, since the timing can impact the executed path. For example, the functional simulator fetches and simulates instructions from the wrong path after a branch has been mispredicted by the timing simulation. When the timing details of the branch simulation determine that its prediction was wrong, it redirects functional simulation on its correct path. All instructions from the wrong path pollute the caches and the internal structures, just as they would do on a normal branch misprediction.

On the other end of the spectrum, *functional-first* (also called *trace-driven*) simulators let the functional simulation produce an open-loop trace of the executed instructions that can later be replayed by a timing simulator. Some trace-driven simulators do not store the instruction traces, but pass them directly to the timing simulator for immediate consumption. A trace-driven approach can only replay what was previously simulated. So, for example, it cannot play the wrong execution path off a branch misprediction, since the instructions trace only contains the correct paths in the execution of the application. To correct for this, timing models normally implement mechanisms to account for the mispredicted execution of instructions, but in a less-accurate way.

The control choice also impacts the engineering architectures of both approaches. Execution-driven simulators are normally programmed with their functional and timing simulators tightly coupled, which makes it easier for the timing to control the functional. Trace-driven simulators are usually built using instrumentation libraries such as Atom [23] or Pin [13], which simplify the functional simulation by running natively in the machine.

As a middle ground, Mauer et al. [14] propose a *timing-first* approach where the timing simulator runs ahead and uses the functional simulator to check (and possibly correct) execution state periodically. This approach clearly favors accuracy vs. speed and was shown to be appropriate for moderately sized multiprocessors and simple applications.

Speed scalability and support for complex benchmarks have become fundamental requirements, and we advocate that another approach is needed, which we call *functional-directed*. Speed requirements mandate that functional simulation should be in the driver's seat and, with sufficient

speed, we can capture larger applications and higher core counts. To overcome the accuracy limitations of traditional trace-based approaches, we add a *timing feedback* mechanism that periodically adjusts the speed of functional emulation to reflect the timing estimates and hence give the running application a more-precise timing correction.

COTSon is the simulation platform jointly developed by HP Labs and AMD that follows this philosophy. In this paper we present its development, capabilities, characteristics and limitations. We describe the mechanisms we developed to accelerate CPU simulation through dynamic sampling techniques. We then describe the network synchronization mechanism that can parallelize simulation of clusters on clusters with a negligible accuracy penalty. We finally present a more forward-looking research use of COTSon in modeling a very high-core-count (1000 cores) hypothetical machine.

2. DESIGN REQUIREMENTS

In this section we describe the design requirements of COTSon. We explain the design decisions we have made. The two factors that most influenced our design decisions were the ability to trade-off accuracy for speed, and the ability to maintain and extend COTSon.

2.1 Decouple functional & timing simulation

COTSon offers decoupled functional and timing simulation by defining a clear interface between both paths. The first benefit of such an approach is the ability to reuse existing functional simulators. These are very complicated to build and maintain, but on the other hand, there are plenty of them being developed and maintained. COTSon's functional simulator uses AMD's SimNowTM simulator¹ [2].

COTSon's decoupled architecture is highly modular. Many interfaces have been designed and built with sufficient generality so that exchanging many of COTSon's functionalities is easy. This modularity enables users to select different timing models, for example, depending on the particular experiment. Moreover, it is easy to program new timing simulators or even adapt existing ones and incorporate them into COTSon. The same is true about other timing models such as network interface cards (NIC) or disk-timing models, and even about simulation techniques such as the sampling strategy being used. This versatility is what makes COTSon a simulation platform.

2.2 Full-system simulation

The SimNow simulator functionally models most of the existing hardware that can be found on an AMD system. Other companies and partners, including HP, develop their own functional devices for the SimNow platform. Full-system simulation means that all aspects of the system can be analyzed by the simulator. All benchmarks that run inside an AMD platform run unmodified in COTSon.

Sometimes difficulties arise when simulating a whole benchmark suite. Some applications use rare and unsupported system calls. Under a full-system simulation this is rarely the case, since everything that runs on the system being

¹SimNow and AMD Opteron are trademarks of Advanced Micro Devices, Inc.

simulated runs on the functional simulator and therefore on the timing simulator.

A full-system simulator also enables the simulation of closed-source, pre-built or legacy applications. For example, we have used COTSon to install, tune, test and simulate Oracle database installations. In order to do so, we gave an expert remote access to a virtual machine environment where he could install and tune our Oracle installation. Afterwards, a snapshot of this virtual machine was repeatedly used for different experiments.

We believe that COTSon’s approach delivers a platform that may be used by many different kinds of users. Network research, usually employing captured or analytically generated traces, may now use COTSon either to generate better traces or to see the impact of their protocols and implementations under real network applications. Other kinds of researchers, such as those interested in storage, OS, microarchitecture, and cache hierarchies may also benefit from COTSon’s holistic approach, enabling the analysis and optimizations of whole systems.

2.3 Faster simulation speed means more data

Even though the speed of the simulation itself does not change the results of the experiments, it is by far one of the most important aspects of a simulation infrastructure. A full-system model implementation may be five or six orders of magnitude slower than the real system. One second of execution of such system takes from one to over ten compute days to simulate. Although several experiments may be run in parallel (with the different parameters to analyze), this approach is unsustainable because it limits the coverage of the experiments.

COTSon is designed with simulation speed as one of its top priorities. COTSon uses virtual machine techniques for its functional simulation, e.g., just-in-time compiling and code caching. The functional simulation is handled by the Sim-Now simulator which has a typical slowdown of $10\times$ with respect to native execution. Other virtual machines such as VMWare [20] or QEMU [4] have smaller slowdowns of around 25%, but their lower functional fidelity and limited range of supported devices make them unsuitable for a full-system simulator.

To speed up timing simulation, we have analyzed both the interfaces with the functional side and all the different device models to simulate. Normally, simulation bottlenecks are in the CPU and memory hierarchy models. To speed these up we have studied the use of simpler models together with parameter-fitting techniques. We have also designed our CPU timing interfaces with sampling in mind, since sampling has been shown to speed up CPU simulation by orders of magnitude.

2.4 Accuracy-vs.-Speed trade-offs

Simulators that target 100% accuracy are extremely expensive in terms of development, validation and testing. Their accuracy normally comes at the expense of other interesting characteristics, such as simulation speed or broadness of the experiments they can carry out, making them too rigid for research purposes.

Speed and accuracy are inversely related, i.e., both cannot be optimized at the same time. We believe that the highest accuracy is not always needed for all kinds of experiments such as initial design space explorations. Many techniques such as sampling replace absolute fidelity with a highly similar behavior on average. COTSon can thus approximate with high confidence the total computing time of a particular application while not knowing its specific behavior at every instant.

COTSon exposes an accuracy-vs.-speed trade-off to its users. This enables them to skip uninteresting parts of the code (such as initial loading) by simulating them at lower accuracy levels. It also allows for fast characterization of benchmarks and systems for their subsequent detailed simulation. All our proposed techniques follow this trade-off philosophy, allowing the user to select, both statically and dynamically, the desired trade-off.

2.5 Timing feedback

Trace-driven timing simulation lacks a communication path from the timing to the functional simulation. The functional simulation is independent of the timing simulation. For many situations, especially unithreaded microarchitecture research, this is not a severe limiting factor. Unfortunately, more complex systems do change their functional behavior depending on their performance.

One example of such a situation occurs in operating systems, which normally use a fixed time quantum to schedule processes and threads. Threads in a multithreaded application also exhibit different interleaving patterns depending on the performance of each of the threads, which in some cases may produce different functional behavior. On another level, many networking libraries such as Message Passing Interface (MPI) change their policies and algorithms depending on the particular performance of the network.

Having *timing feedback*—a communication path from the timing to the functional simulator—is crucial for studying these kinds of situations. Since COTSon aims to be used as a full-system simulator, pure trace-driven simulation cannot be considered as an option. On the other hand, execution-driven approaches with timing feedback are inherently very complicated and hard to speed up using current virtual machine techniques. Instead of choosing any of these fixed extremes, COTSon follows a middle road, by combining sampled trace-driven approaches with timing feedback.

COTSon makes its functional simulator run for a dynamically set interval. This produces a stream of events which are sent to the respective CPU timing models. At the end of the interval, using the metrics from the CPU models (which include the whole memory hierarchy) a new IPC (instructions per cycle) is fed back to the functional simulator. By selecting different interval sizes, the user can turn the accuracy-vs.-speed knob either way. Moreover, COTSon’s approach couples very well with sampling, enabling the user to select just those intervals which are considered representative or “interesting.” Another benefit is the capability of trace generation for off-line simulation on those occasions in which trace-driven simulation is the selected path.

2.6 Scaling up and out

The very first suggestion from Yi et al. [30] is to enable multicore simulation. COTSon's functional simulator, AMD's SimNow, is one of the few commercially available virtual machines with an important commitment to multicore functional simulation. COTSon currently supports simulation of all existing AMD systems. In order to study future systems, it is important to be able to simulate an unbounded number of cores in a system. Simulating dozens or hundreds of cores is not trivial, since it implies changes in the simulator, the BIOS, and the OS.

Scaling out is also among COTSon features. Currently, over 75% of all systems in the Top 500 list [26] are clusters, and they have become the de facto commodity option for both high performance and scalability. COTSon's approach to simulating a cluster is radically different from previous approaches for simulating parallel machines. COTSon combines individual node simulators to form a cluster simulator.

3. COTSON'S ARCHITECTURE

Figure 1 shows an overview of the COTSon architecture. COTSon uses AMD's SimNow simulator for the functional simulation of each node in the cluster. The SimNow simulator is a fast and configurable x86 and AMD64 platforms simulator for AMD's family of processors. It uses dynamic compilation and caching techniques to speed up CPU simulation. It provides an accurate model of a computer system from the program, OS, and programmer point of view. It is capable of booting a system with an unmodified BIOS and OS, and executes any kind of complex application. The SimNow simulator is able to simulate both uniprocessor and multiprocessor systems, and includes full simulation of system devices. The SimNow platform has an SDK available to partners where new devices can be programmed that mimic real hardware.

In order to add timing simulation to COTSon, HP Labs and AMD have jointly augmented the SimNow simulator with a double communication layer which allows any device to export functional events and receive timing information from them. All events are directed by COTSon to their timing model, which is selected by the user. Each timing model may describe which events it is interested in via a dynamic subscription mechanism. There are two main types of device communication: synchronous and asynchronous.

3.1 Synchronous simulation

Synchronous communication devices are those devices that immediately respond with timing information for each event received. Currently COTSon supports and has different timing models for synchronous devices such as disks and NICs.

One example of synchronous communication is the simulation of a disk read by the functional simulator. The IDE device is responsible for handling the read operation, finding out the requested data and making it available to the functional simulator. However, instead of issuing the equivalent of an interrupt, the functional simulator issues a read event with all the pertinent information to COTSon. COTSon delivers this event to a specific disk model which determines its latency. The latency is used by the SimNow simulator to

schedule the functional interrupt that signals the completion of the read.

3.2 Asynchronous simulation

A synchronous simulation approach is not viable for events that happen at a high frequency. If each instruction executed by the functional simulator had to communicate with a timing model, the simulation would come to a halt. Virtual machines benefit extraordinarily from caching the translations of the code they are simulating, and staying inside the code cache for as long as possible. A synchronous approach would mean that the simulation of each instruction would have to leave the code cache, imposing a very big penalty. On top of this, current CPU timing models are unable to estimate the speed of each instruction by itself. Superscalar microarchitectures have many instructions executing at the same time, thus the concept of instruction latency must be substituted with the average IPC.

Asynchronous communication is needed for these kinds of devices. Asynchronous communication decouples the generation of events and its processing and the feedback of timing information. Instead of receiving a call per event, the SimNow simulator produces tokens describing dynamic events. These tokens are parsed by COTSon and delivered to the appropriate timing modules. At specific moments, COTSon asks these timing modules for aggregate timing information (in the form of IPC) and feeds it back to each of the functional cores.

3.3 Asynchronous timing feedback interface

The IPC fed back into the functional cores is used by the SimNow simulator to schedule the progress of instructions in each core. Whenever COTSon provides a new IPC value, the scenario in which applications and OS are being simulated changes. The simulated system time evolves based on the output from the timing modules.

Unfortunately, just programming the functional IPC with the results of the CPU timing modules is not a good idea. There are many situations in which the information from the CPU timing modules has to be filtered and adapted before passing it to the SimNow simulator. An example of this occurs with small samples. If a particular core is mostly idle, the number of instructions in that sample may not be enough to get an accurate estimate of the IPC. Feeding back the resulting IPC may reduce simulation accuracy.

Instead of forcing timing modules to implement all possible corrections needed, COTSon offers a timing feedback interface that handles such corrections transparently to the timing modules. One of the most interesting features of this interface is the potential to correct and accurately predict future IPC by using mathematical models such as Auto-Regressive-Moving-Average (ARMA) model, which is borrowed from the field of time-series forecasting.

3.4 Multicore instruction interleaving

AMD's SimNow simulator is able to simulate multicore architectures with a moderate number of cores, by "sequentially" the execution of the different cores. Each core is allowed to run independently for some maximum amount of

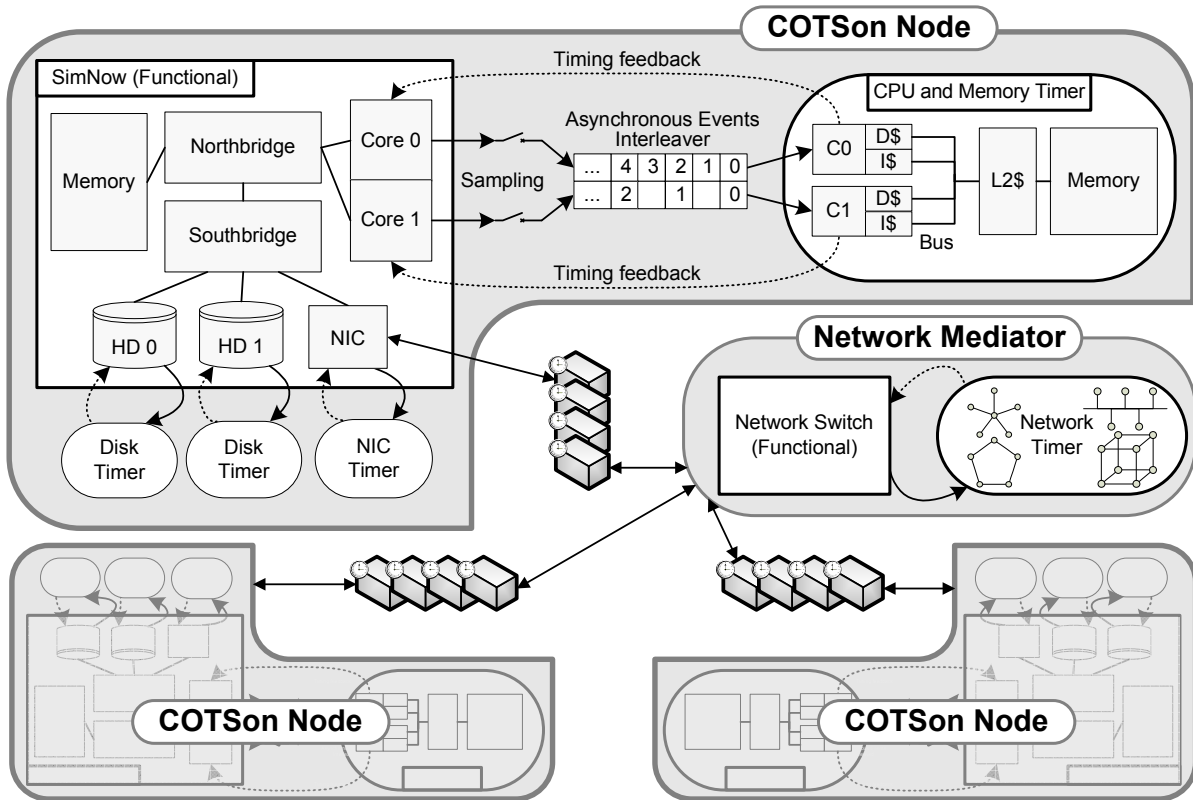


Figure 1: COTSon’s architecture

time, called multiprocessor synchronization quantum, which can be programmed by COTSon. At the end of the synchronization quantum, all the cores have reached the same point in time.

The simulation of each core generates a series of events that are stored into an asynchronous queue, following the methodology described in Section 3.2. COTSon parses these queues to build higher-level objects such as instructions. Instead of sending the instructions or memory accesses immediately to the timing models, COTSon waits until it has available instructions from all the cores and interleaves them based on how the CPU timing models consider appropriate.

In most cases, the interleaving of instructions by the timing modules differs from what the SimNow simulator has executed, but this difference only impacts the perceived performance of the application, and COTSon already handles that through timing feedback. There are functional differences that COTSon does not capture this way. An example of this is an active wait by a spin lock. The functional simulator may decide to spin five iterations before the lock is acquired, while the timing simulator, depending on the model being used, may determine that it should have iterated 10 times or none. This discrepancy impacts the accuracy of the simulation. Nevertheless, COTSon provides a mechanism, the multiprocessor synchronization quantum, which increases accuracy at the expense of a lower simulation speed. Another possibility we have followed for certain experiments consists of tagging all fine-grain synchronization mechanisms in the

application code being analyzed. When these tags arrive at the interleaver, COTSon is able to simulate the pertinent instructions based on timing information. Tagging synchronization primitives offers the highest-possible simulation accuracy.

3.5 Sampling

Asynchronous simulation of CPU cores is extremely slow. The SimNow simulator has a typical slowdown of $10\times$, which means that on average the execution of 1 native instruction takes 10 instructions in the code cache. Unfortunately, timing simulation of one instruction may have a slowdown between $10,000\times$ and $100,000\times$. The functional simulation inserts extra code in the code cache that dynamically generates events for the timing modules. The cycle-accurate simulation of these events, namely instructions and memory accesses, involves fetching the instruction, decoding it, renaming it, and so on.

Accurate timing simulation of a CPU core and memory hierarchy is extremely slow compared to its functional simulation. It is so slow that coupled cycle-accurate simulators rarely try to optimize their functional simulation. After all, it represents just a minuscule part of their total execution.

The best way of speeding up timing simulation is considered to be sampling [31]. Sampling consists of determining what are the interesting or representative phases of the simulation and just simulating those. The results from these samples are then combined to produce global results.

Sampling is central to asynchronous devices. The sampler is selected by the user per experiment and is responsible for deciding the representative phases of execution. It does so by selecting the type of the current sample and its length, i.e., COTSon asks the sampler what to do next and for how long. The sampler may reply with a command to enter one of four distinct phases: *functional*, *simple warming*, *detailed warming* and *simulation*.

In the *functional* phase, an asynchronous device does not produce any kind of events, and so runs at full speed. The *simulation* phase is the opposite: COTSon instructs the device to produce events and sends them to the timing models. In order to remove the non-sampling bias from the simulation, most samplers require that the timing models be warmed up. COTSon understands two different warming phases: *simple warming* is intended for warming the high-hysteresis elements, such as caches and branch target buffers; *detailed warming* is intended for warming up both high-hysteresis elements and also low-hysteresis ones, such as reorder buffers and renaming tables. Normally the sampler inserts one or several warming phases before switching to simulation.

The sampler may be controlled from inside the functional system via the use of special libraries and utilities. Users may use them to annotate their applications which then send finer-grain phase selection information to the sampler.

3.6 External networking

COTSon supports a NIC device. NIC devices are very simple and merely determine how long a particular network packet takes to be processed by the hardware.

When a particular application needs to communicate using the network, it executes some code that eventually reaches the NIC. This produces a NIC event that reaches the NIC synchronous timing model. The response time from the timing model is then used by the functional simulator to schedule the emission of the packet into the external world. The packet is sent to an external entity which is called the network mediator. Among the functionalities of the mediator are those that allow the packet to reach the external world. It also offers a Dynamic Host Configuration Protocol (DHCP) for the virtual machines and allows the redirection of ports from the host machine into the guest simulated system.

3.7 Cluster networking

In order to simulate a whole cluster, COTSon instantiates several COTSon node simulators, potentially in different host machines. Each of them is a stand-alone application which communicates with the rest via the network mediator. The network mediator acts as a functional network switch, directing network packets to the appropriate COTSon node destination.

To simulate network latencies, the mediator relies on network timing models, which determine the total latency of each packet based on its characteristics and the network topology. The latency is attached to each packet and is used by the destination COTSon node simulator to schedule the arrival of the packet.

The mediator is also responsible for the time synchronization of all the node instances. Without synchronization, each of the nodes simulated would see time advance at a different rate. In real life this would be similar to assuming all nodes have skewed system clocks which work at different frequencies. Although this does not prevent most cluster applications from completing successfully, COTSon is unable to report objective measurements from the timing simulation. COTSon uses the mediator to synchronize all node simulators and forces them to proceed in a controlled way.

4. RESEARCH CHALLENGES

The COTSon platform has involved an extraordinary amount of development on both HP's and AMD's side. But not everything has been development; many of the problems we have encountered were new and needed research. In this section, we present three research problems solved successfully by HP Labs, as well as the impact that this research has had on COTSon.

4.1 Sampling to speed up CPU simulation

One of the first analyses done to COTSon revealed that the biggest bottleneck to simulation performance was CPU simulation. The frequency of CPU events is several orders of magnitude bigger than that of any other kind of device. It was clear that optimizing simulation performance came through optimizing CPU simulation.

Sampling techniques [11] selectively turn on and off timing simulation, and are among the most promising for improving timing simulation. Other techniques, such as using a reduced input set or simulating just an initial portion of programs, also reduce simulation time, but at the expense of accuracy. Sampling is the process of selecting appropriate simulation intervals, so that the extrapolation of the simulation statistics in these intervals well approximates the statistics of the complete execution. Previous work has shown that an adequate sampler can yield excellent simulation accuracy. The two most cited samplers for microarchitectural simulation are SimPoint [22] and SMARTS [29].

4.1.1 SMARTS

SMARTS employs systematic sampling. It makes use of statistical analysis in order to determine the number of instructions that need to be simulated in the desired benchmark (number of samples and length of samples). As simulation samples in SMARTS are rather small (~ 1000 instructions) it is crucial for SMARTS to keep micro-architectural structures such as caches and branch predictors warmed-up all the time. For this reason, they perform a simple warming (functional warming in SMARTS parlance) between samples. The statistical analysis consists of running a previous profiling phase, i.e., a complete execution of the application, to collect data and then determine the most appropriate simulation points. This analysis can be done once per benchmark and then reused for many experiments, as long as the functional simulation of the benchmarks is not affected by the timing simulation, i.e., no timing feedback is ever allowed under SMARTS.

4.1.2 SimPoint

SimPoint also needs a full profile of the benchmarks. The obtained profile of instructions is later processed using clustering and distance algorithms to determine the best simulation points. As with SMARTS, this off-line analysis works just for this benchmark and input set, and must be redone if any of the characteristics of the application changes. The first SimPoint step performs an *a priori* static profiling of the code being executed to create code signatures that represent the program’s behavior at different points during execution. These signatures are then used to concisely determine which portion of the code is best to simulate. Recent versions of SimPoint include clustering analysis and other optimizations, such as multiple simulation intervals and variable-length intervals, that improve SimPoint’s speed and accuracy [5]. The simulation points obtained by SimPoint are used to gain an understanding of whole program behavior and to greatly reduce simulation time by using only representative samples.

In both SimPoint and SMARTS, the number and duration of the full-timing simulation samples determine the total simulation time and simulation accuracy. Results found in the literature show that SMARTS is generally more accurate than SimPoint. However, as the sampling selection mechanism in SimPoint is faster, SimPoint offers the best accuracy-vs.-speed trade-off [31].

4.1.3 Problems of existing samplers

Unfortunately, samplers like SMARTS or SimPoint assume a stable and repeatable functional execution regardless of the timing. In these scenarios, the off-line processing is accurate and can determine the most representative parts of the application as samples. In the presence of timing feedback, these samplers cannot rely anymore on their sample selection process. As we saw earlier, timing feedback is fundamental for OS and multithreaded application analysis. Moreover, SMARTS imposes a very heavy load, requiring dynamic information for each instruction, preventing COTSon from ever running in the *functional* phase which produces the biggest speed improvement.

4.1.4 Dynamic Sampling

Dynamic Sampling [7] dynamically adapts the timing simulation to the application characteristics (where application includes the full system simulation). This approach has two fundamental advantages: 1) it frees COTSon from the profiling phase needed by both SMARTS and SimPoint, and 2) it allows the experimenter to dynamically adjust the accuracy-vs.-speed knob depending on the characteristics of the application and the particular experiment being done.

The basis for Dynamic Sampling lies in the following: in the process of simulating a complete system, a functional simulator performs many different tasks and keeps track of several statistics. The SimNow simulator works more or less the same; it maintains a series of internal statistics collected during the simulation. These statistics measure elements of the simulated system as well as the behavior of its internal structures. The statistics related to the characteristics of the simulated code are similar to those collected by microprocessor hardware counters. For example, the SimNow simulator maintains, among other statistics, the number of executed

instructions, memory accesses, exceptions, and bytes read or written to or from a device. This data is inherent to the simulated software and is a clear indicator of the behavior of the running applications. The correlation of changes in code locality with overall performance is a property that other researchers have already established [12].

The SimNow simulator also keeps track of statistics of its internal structures, such as the translation cache and the software translation lookaside buffer (TLB, necessary for efficient implementation of simulated virtual memory). Intuitively, one can imagine that this second class of statistics could also be useful to detect phase changes of the simulated code. Our results show that this is indeed the case.

Dynamic sampling uses a sensitivity value that indicates the minimum first-derivative threshold of the monitored variable that triggers a phase change. The control logic of our algorithm inspects the monitored variables at regular intervals. Whenever the relative change between successive measurements is larger than the sensitivity, it triggers a new simulation interval with full timing.

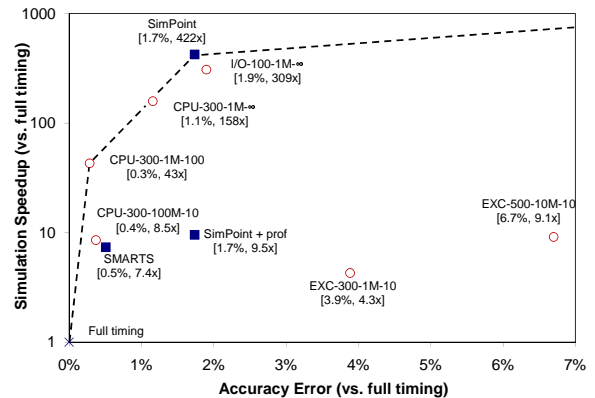


Figure 2: Dynamic sampling results

Figure 2 shows a summary description of the speed vs. accuracy trade-offs of the proposed Dynamic Sampling approach and how it compares with SMARTS and SimPoint sampling techniques. In these experiments, we simulate the whole SPEC CPU2000 benchmark suite [24] using the reference input until completion or until they reach 240 billion instructions, whichever occurs first.

On the x axis we plot the accuracy error versus what we obtain in a full-timing run (smaller is better). On the logarithmic y axis we plot the simulation execution speedup versus the full-timing run (larger is better). Each point represents the accuracy error and speed of a given experiment averaged for all the benchmarks, all normalized to a full timing run (speed=1, accuracy error=0). The graph shows four square points: full timing, SMARTS, and SimPoint with and without considering profiling and clustering time. Circular points are some interesting results of Dynamic Sampling, with various configuration parameters. The terminology used for these points is “AA-BB-CC-DD”, where AA is the SimNow internal statistic that is monitored (CPU for code cache invalidations, EXC for code exceptions, and I/O for I/O operations), BB is the sensitivity value that determines

phase change, CC is the interval length (in number of instructions), and DD is the maximum number of consecutive functional intervals that can exist between two consecutive simulation intervals.

The dotted line shows the *Pareto optimality curve* highlighting the “optimal” points of the explored space. A point in the figure is considered Pareto optimal if there is no other point that performs at least as well on one criterion (accuracy error or simulation speedup) and strictly better on the other criterion. Note that both SMARTS and SimPoint are in (or very close to) the Pareto optimality curve, which implies that they provide two very good solutions for trading accuracy vs. speed.

The four Dynamic Sampling results in the left part of the graph are particularly interesting. These reach accuracy errors below 2%, and as little as 0.3% (in “CPU-300-1M-100”). The difference between these points is in the speedup they obtain, ranging from $8.5\times$ (similar to SMARTS) to an impressive $309\times$. An intermediate point with a very good accuracy/speed trade-off is “CPU-300-1M- ∞ ”, with an accuracy error of 1.1% and a speedup of $158\times$.

4.2 Scaling out: Cluster simulation

COTSon distributes the simulation of the different cores over multiple hosts. Synchronizing these COTSon node instances can be accomplished using Parallel Discrete Event Simulation (PDES) techniques [9, 15], all of which basically require simulation to synchronize at given intervals, called *quanta*. Unfortunately, doing so in a straightforward way implies forcing very small synchronization quanta, smaller than the shortest simulated network latency. For modern low-latency fabrics with round-trips of few microseconds, the overhead of perfect synchronization would cause about two orders of magnitude slowdown in cluster simulations.

We have implemented an adaptive quantum synchronization technique [8] that follows COTSon’s accuracy-vs.-speed trade-off philosophy. Since applications are not always sending packets, they do not need to work at the smallest synchronization quantum during intervals where communication is sparse. Thus with this technique, in the absence of packets, simulation accelerates to the highest possible performance by increasing the granularity of synchronization. As soon as packet traffic increases, simulation quickly decelerates to the smallest synchronization quantum, to retain simulation accuracy.

As nodes advance at different speeds, it may happen that some packets reach their destination when this has already passed their delivery time. In these situations COTSon delivers these *straggler packets* immediately, but some accuracy is lost due to the late delivery. Since the number of straggler packets is kept small, COTSon manages to keep the total impact on accuracy bounded, while still speeding up simulation considerably. However, the price to pay is making simulation non-deterministic.

Figure 3 shows a summary description of the speed vs. accuracy trade-offs of two configurations of adaptive quantum synchronization and how they compare with the experiments run with bigger quanta. The results shown in this figure cor-

respond to a cluster with 8 nodes. We present two sets of results, a first one using the NAS benchmark suite [17], and a second one using the NAMD benchmark [19].

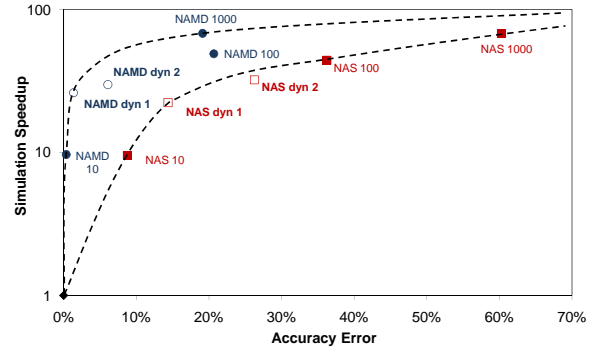


Figure 3: Adaptive quantum results

On the x axis we plot the relative accuracy error, as a percentage, with respect to the baseline, $quantum = 1\mu s$, thus smaller values represent better accuracy. On the logarithmic y axis we plot the simulation execution speedup with respect to the same baseline. Each point represents the accuracy error and speed of a given experiment. Our two configurations have been tagged as *dyn 1* (using a 3% acceleration factor) and *dyn 2* (5% acceleration).

The dotted lines show the *Pareto optimality curve* for the two benchmark suites, highlighting the “optimal” points of the explored space. All adaptive configurations lie in or very near the Pareto curve, and can thus be considered nearly optimal. COTSon’s novel approach to cluster simulation achieves an order of magnitude faster cluster experiments while maintaining acceptable accuracy.

4.3 Scaling up: Manycore simulation

The obvious way of scaling up a full system simulator is to adapt its functional simulation to enable a larger number of functional cores. Although COTSon is already working on this aspect, its applicability is not immediate, since both the BIOS and the OS need substantial changes that enable them to manage a large number of cores in an efficient way.

In parallel with the more general approach, COTSon has implemented a novel technique [16], which — although more limited in its applicability — shows great potential for a quick study of several high-performance computing benchmark suites. COTSon converts time-multiplexed threads into space-multiplexed threads, by transforming software threads that were previously mapped to hardware threads into threads mapped to each of the cores. We believe that this approach, which has already been used successfully in several research papers [25, 27], is an important first step towards the simulation of chip multiprocessors and future manycore architectures.

COTSon augments the functional simulator to identify the instruction streams. Instructions corresponding to different software threads are detected and tagged. We manually modify the source code of the OS (specifically, a simple call inside the *context switch* routine), which enables COTSon

to detect task changes in the system and determine which threads and processes are executing at each moment.

Once instructions from the different software threads are tagged, COTSon dynamically maps each instruction flow into the corresponding hardware core of the simulated many-core architecture. The inherent thread synchronization is also taken into account by annotating the synchronization points of the application. COTSon isolates synchronous operations such as locks and barriers, and maps them onto the target manycore architecture. COTSon uses annotation to correctly simulate them and also to remove the effects that certain implementations, such as spinning locks, may have on the functional simulation from which they are derived. Discarding these effects, and others such as I/O device simulation and certain parts of the OS, is crucial since they are no longer meaningful under the target architecture.

COTSon enhances its interleaving mechanism to capture these annotations and separate software threads and direct them into the different models for the cores. The fact that the interleaver needs instructions from all the cores to proceed poses an interesting problem. If the underlying OS decides not to schedule a thread, COTSon has to store all the instructions from other threads until all threads are scheduled. This causes an important scalability issue: the amount of memory depends on OS scheduling and can quickly grow out of control. To circumvent this problem, COTSon feeds back scheduling information to raise the priority of software threads with a low number of instructions.

Figure 4 shows the accumulated IPC (throughput) over time of two SPLASH-2 [28] benchmarks, FFT and Barnes, executing in a manycore machine with a varying number of cores (64–1024 cores). Each line in the graph corresponds to a different number of cores; the y axis shows accumulated IPC, while the x axis shows number of cycles (note that executions with more nodes take fewer to complete). In these simulations, COTSon simulates an ideal machine, with each core having an in-order pipeline and a constant-latency memory system. These experiments are thus useful to understand the intrinsic scaling limitations of an application due to synchronization and dataset. As shown in Figure 4, FFT scales almost linearly, achieving the maximum IPC for most of its execution time. On the other hand, Barnes shows some phases that do not scale at all, lasting almost the same regardless the number of cores. These phases (*building tree*) are the main bottleneck of this application and prevent it from achieving peak performance for a high core count.

5. CONCLUSIONS

The COTSon project started in 2006 with the aim of building a simulation infrastructure that could predict the performance of existing and future computer systems. At that time, there was no tool that could measure the performance of a full system composed of hundreds of multicore, multiprocessor nodes, including the full software stack and all the system devices like network cards or disks, in an affordable amount of time.

Today COTSon is a mature platform whose adoption is growing inside and outside HP Labs. Due to its capabilities to dynamically adjust accuracy and speed, COTSon can be

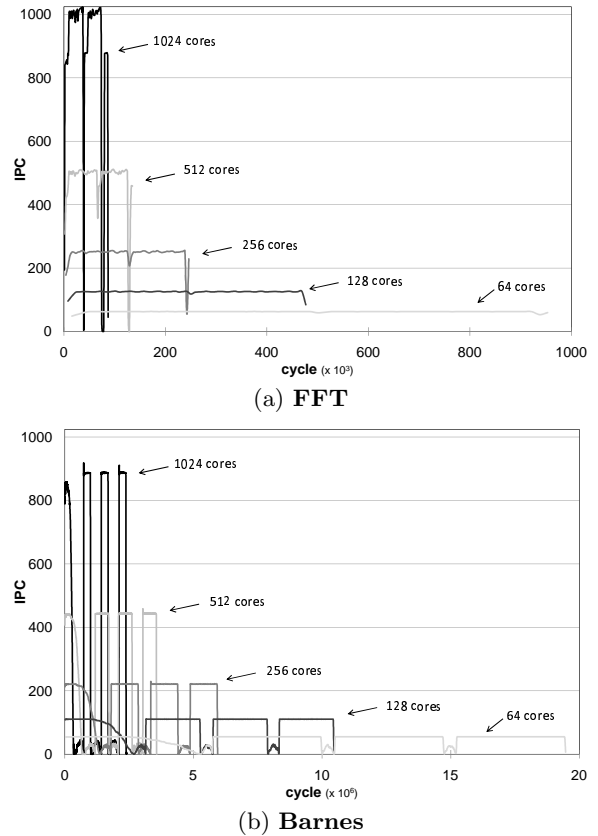


Figure 4: IPC over time for two SPLASH-2 benchmarks and number of cores ranging from 64 to 1024

used at very different levels. On the one hand, at a microarchitectural level, it can be used by researchers in computer architecture wanting to measure the performance of their new branch predictor or cache prefetching algorithm. On the other hand, at the system level, COTSon can be used as a bid-desk tool to optimize the combination of computing resources to match customer requirements in terms of performance/power and cost.

We plan to extend COTSon in several ways. We have developed an SDK so that COTSon users can incorporate their CPU timing and power models into the simulation infrastructure. As COTSon gets more widely used by the research community, we expect that many new features will be included in the existing infrastructure, such as reliability measurement tools or profiling and visualization tools.

Acknowledgments

We thank the AMD SimNow team for their contributions to the COTSon infrastructure.

6. REFERENCES

- [1] Ambric. Massively Parallel Processor Array technology. <http://www.ambric.com>.
- [2] R. Bedicheck. SimNow: Fast platform simulation purely in software. In *Hot Chips 16*, Aug. 2004.
- [3] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, and M. Reif. TILE64 processor:

- A 64-core SoC with mesh interconnect. In *Proceedings of the International Solid-State Circuits Conference (ISSCC 2008)*, Feb. 2008.
- [4] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX 2005 Annual Technical Conf., FREENIX Track*, pages 41–46, Apr. 2005.
- [5] B. Calder. SimPoint. <http://www.cse.ucsd.edu/~calder/simpoint>.
- [6] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar. An integrated quad-core Opteron processor. In *IEEE International Solid-State Circuits Conference (ISSCC 2007)*, Feb. 2007.
- [7] A. Falcón, P. Faraboschi, and D. Ortega. Combining simulation and virtualization through dynamic sampling. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software*, Apr. 2007.
- [8] A. Falcón, P. Faraboschi, and D. Ortega. An adaptive synchronization technique for parallel simulation of networked clusters. In *Proc. of the 2008 IEEE International Symp. on Performance Analysis of Systems & Software*, Apr. 2008.
- [9] R. M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, 1990.
- [10] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell’s multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.
- [11] T. Lafage and A. Sez nec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. *Workload Characterization of Emerging Computer applications*, pages 145–163, 2001.
- [12] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *Proceedings of the Intl. Symposium on Performance Analysis of Systems and Software*, pages 236–247, Mar. 2005.
- [13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2005.
- [14] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-system timing-first simulation. In *SIGMETRICS ’02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 108–116, New York, NY, USA, 2002. ACM.
- [15] J. Misra. Distributed discrete-event simulation. *ACM Comput. Surv.*, 18(1):39–65, 1986.
- [16] M. Monchiero, J.-H. Ahn, A. Falcón, D. Ortega, and P. Faraboschi. How to simulate 1000 cores. In *Workshop on Design, Architecture and Simulation of Chip Multiprocessors (dasCMP’08)*, Nov. 2008.
- [17] NASA Ames Research Center. The NAS parallel benchmarks. <http://www.nas.nasa.gov/Resources/Software/npb.html>.
- [18] U. G. Nawathe, M. Hassan, L. Warriner, K. Yen, B. Upputuri, D. Greenhill, A. Kumar, and H. Park. An 8-core 64-thread 64-bit power efficient SPARC SoC (Niagara2). In *Proceedings of the International Solid-State Circuits Conference (ISSCC 2007)*, pages 108–109, 2007.
- [19] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, Oct. 2005.
- [20] M. Rosenblum. VMware’s virtual platform: A virtual machine monitor for commodity PCs. In *Hot Chips 11*, Aug. 1999.
- [21] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel Distrib. Technol.*, 3(4):34–43, 1995.
- [22] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, Oct. 2002.
- [23] A. Srivastava and A. Eustace. ATOM — a system for building customized program analysis tools. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 1994.
- [24] Standard Performance Evaluation Corporation. SPEC CPU2000. <http://www.spec.org/cpu2000>.
- [25] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *Proc. of the 35th Annual International Symposium on Computer Architecture*, June 2008.
- [26] TOP500 Project. TOP500 Supercomputer Sites. <http://www.top500.org>.
- [27] D. Vantrease, R. Schreiber, M. Monchiero, M. McLaren, N. P. Jouppi, M. Fiorentino, A. Davis, N. Binkert, R. G. Beausoleil, and J. H. Ahn. Corona: System implications of emerging nanophotonic technology. In *ISCA ’08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 153–164, 2008.
- [28] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.
- [29] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual Intl. Symposium on Computer Architecture*, pages 84–97, June 2003.
- [30] J. J. Yi, L. Eeckhout, D. J. Lilja, B. Calder, L. K. John, and J. E. Smith. The future of simulation: A field of dreams. *Computer*, 39(11):22–29, 2006.
- [31] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and comparing prevailing simulation techniques. In *Proceedings of the 11th Intl. Conference on High Performance Computer Architecture*, pages 266–277, Feb. 2005.