

Combining Simulation and Virtualization through Dynamic Sampling

Ayose Falcón

Paolo Faraboschi

Daniel Ortega

Hewlett-Packard Laboratories

Advanced Architecture Lab — Barcelona Research Office

{ayose.falcon, paolo.faraboschi, daniel.ortega}@hp.com

Abstract

The high speed and faithfulness of state-of-the-art Virtual Machines (VMs) make them the ideal front-end for a system simulation framework. However, VMs only emulate the functional behavior and just provide the minimal timing for the system to run correctly. In a simulation framework supporting the exploration of different configurations, a timing backend is still necessary to accurately determine the performance of the simulated target.

As it has been extensively researched, sampling is an excellent approach for fast timing simulation. However, existing sampling mechanisms require capturing information for every instruction and memory access. Hence, coupling a standard sampling technique to a VM implies disabling most of the “tricks” used by a VM to accelerate execution, such as the caching and linking of dynamically compiled code. Without code caching, the performance of a VM is severely impacted.

In this paper we present a novel dynamic sampling mechanism that overcomes this problem and enables the use of VMs for timing simulation. By making use of the internal information collected by the VM during functional simulation, we can quickly assess important characteristics of the simulated applications (such as phase changes), and activate or deactivate the timing simulation accordingly. This allows us to run unmodified OS and applications over emulated hardware at near-native speed, yet providing a way to insert timing measurements that yield a final accuracy similar to state-of-the-art sampling methods.

1. Introduction

Simulators are widely used to assess the value of new proposals in Computer Architecture. Simulation allows researchers to create a virtual system in which new hardware components can be shaped, and architectural structures can

be combined to create new functional units, caches, or entire microprocessor systems.

There are two components in a typical computer simulation: functional and timing simulation. *Functional simulation* is necessary to verify correctness. It emulates the behavior of a real machine running a particular OS and models common devices like disks, video, or network interfaces. *Timing simulation* is used to assess the performance. It models the operation latency of devices emulated by the functional simulator and assures that events generated by these devices are simulated in a correct time ordering.

More recently, power simulation has also become important, especially when analyzing datacenter-level energy costs. As with timing simulation, a functional simulation is in charge of providing events from CPU and devices, to which we can apply a power model to estimate the overall system consumption.

Trace-driven simulators decrease total simulation time by reducing the functional simulation overhead. They employ a functional simulator to execute the target application once, save a trace of interesting events, and then repeatedly use the stored event trace with different timing models to estimate performance (or power). A severe limitation of trace-driven simulation is the impossibility to provide timing-dependent feedback to the application behavior. For this reason, trace-driven approaches that work for uniprocessor single-threaded application domain are less appropriate for complete system simulation. In many cases, the behavior of a system directly depends on the simulated time of the different events. For example, many multithreaded libraries use active wait loops because of the performance advantage in short waits. Network protocols may re-send packets depending on the congestion of the system. In these, and many others, scenarios feedback is fundamental for an accurate system simulation.

Execution-driven simulators directly couple the execution of a functional simulator with the timing models. However, traditional execution-driven simulation is several orders of magnitude slower than native hardware execution, due to the overhead caused by applying timing simula-

tion for each instruction emulated. For example, let's consider the SimpleScalar toolkit [1], a commonly used execution-driven architectural simulator. The typical execution speed of pure functional simulation (*sim-fast* mode in SimpleScalar) is around 6–7 million simulated Instructions Per Second (MIPS) on a modern simulation host capable of 1,000–2,000 MIPS. Hence, we have a slowdown of 2–3 orders of magnitude. If we add timing simulation (*sim-outorder* mode in SimpleScalar), the speed drops dramatically to ~ 0.3 MIPS, that is another 1–2 orders of magnitude. Adding all up, a timing simulation can easily be 10,000 times slower than native execution (i.e., 1 minute of execution in ~ 160 hours). In practice, this overhead seriously constrains the applicability of traditional execution-driven simulation tools to simple scenarios of single-threaded applications running on a uniprocessor system.

Researchers have proposed several techniques to overcome the problem of execution-driven simulators, by improving both the functional and the timing component. Sampling techniques [11, 15, 21] selectively turn on and off timing simulation, and are among the most promising for accelerating 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 much lower 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 sampling methodology can yield excellent simulation accuracy.

However, sampling only helps the timing simulation. Functional simulation still needs to be performed for the entire execution, either together with the timing simulation [21] or off-line, during a separate application characterization (profiling) pass [15]. This characterization phase, which consists in detecting representative application phases, is much simpler than full timing simulation, but still adds significant extra time (and complexity) to the simulation process. In all these cases, simulation time is dominated by the functional simulation phase, which can take several days even for a simple uniprocessor benchmark [21].

In recent years *virtualization* techniques have reached full maturity: modern Virtual Machines (VMs) are able to faithfully emulate entire systems (including OS, peripherals and complex applications), at near-native speed. Ideally, this makes them the perfect candidate for functional simulation.

In this paper we advocate a novel approach that combines the advantages of fast emulators and VMs with the timing accuracy of architectural simulators. We propose and analyze a sampling mechanism for timing simulation that enables coupling a VM front-end to a timing back-end

by minimizing the overhead of exchanging events at their interface. The proposed sampling mechanism is integrated in an execution-driven platform and does not rely on previous profiling runs of the system, since this would be inappropriate for complete systems simulation requiring timely feedback. To identify the appropriate samples, we propose a mechanism to dynamically detect the representative timing intervals through the analysis of metrics that are available to the VM without interrupting its normal execution. This allows us to detect program phases at run time and enable timing simulation only when needed, while running at full speed during the remainder of the execution.

2. Related Work

The search for an optimal combination of accurate timing and fast simulation is not new. However, the majority of authors have focused on improving functional and timing simulation as separate entities; few have proposed solutions to combine the best of both worlds. In this section we review some of the techniques that enhance functional simulation and timing simulation separately. Finally, we review proposals that combine timing simulation with fast functional simulation.

2.1. Accelerating Timing Simulation

The most promising mechanisms to speed up timing simulation are based on sampling. SimPoint [7, 15] and SMARTS [21] are two of the most widely used and referenced techniques. Both represent different solutions to the problem of selecting a sample set that is representative of a larger execution. While SimPoint uses program-phase analysis, SMARTS uses statistical analysis to obtain the best simulation samples. If the selection is correct, we can limit simulation to this sample set, and obtain results that are highly correlated to simulating the complete execution. This dramatically reduces the total simulation time.

It is important to observe that existing sampling mechanisms reduce the overhead due to timing simulation, but still require a complete “standard” functional simulation. Sampling mechanisms rely upon some information of each and every instruction emulated, like its address (PC), its operation type, or the generated memory references. This information is used to detect representative phases and to warm-up stateful simulation structures (such as caches, TLBs, and branch predictors).

However, if our goal is to simulate long-running applications, functional simulation quickly becomes the real speed bottleneck of the simulation. Besides, off-line or *a priori* phase detection is incompatible with timing feedback, which on the other hand is necessary for complete system simulation as we discussed in the introduction.

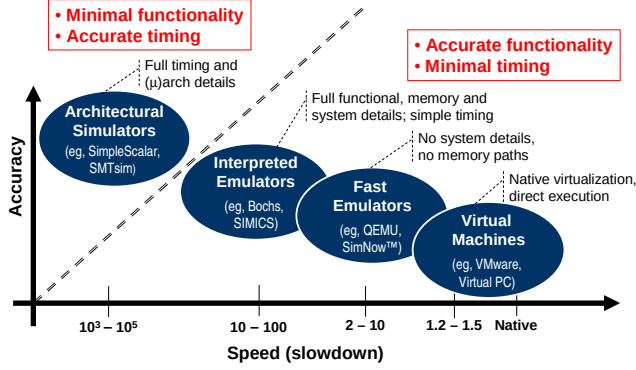


Figure 1. Accuracy vs. Speed of some existing simulation technologies

Architectural simulators like SimpleScalar [1] (and its derivatives), SMTsim [17] or Simics [13] employ a very simple technique for functional simulation. They normally employ *interpreted* techniques to fetch, decode and execute the instructions of the target (simulated) system, and translate their functionality into the host ISA. The overhead of the interpreter loop is significant and is what primarily contributes to limit the functional speed of an architectural simulator. This adds a severe performance penalty in the global simulation process, and minimizes the benefits obtained by improving timing simulation.

2.2. Accelerating Functional Simulation

Several approaches have been proposed to reduce the functional simulation overhead in simulators that use interpretation. By periodically storing checkpoints of functional state of previous functional simulation, some proposals transform part of the execution-driven simulation into trace-driven simulation [18, 19]. The overhead of functional simulation is effectively reduced, but at the expense of creating and storing checkpointing data. What is worse is that checkpointing techniques, as any other off-line technique, also inhibits timing feedback.

Virtualization techniques open new possibilities for speeding up functional simulation. Figure 1 shows how several virtualization emulators and VM technologies relate to one another with respect to timing accuracy and execution speed. Other taxonomies for VMs —according to several criteria— have been proposed [16, 20], which are perfectly compatible with the classification provided in this paper.

Fast emulators and VMs make use of dynamic compilation techniques, code caching and linking of code fragments in the code cache to accelerate performance, at the expense of system observability. These techniques dynamically translate sequences of target instructions into func-

tionally equivalent sequences of instructions of the host. Generated code can be optionally optimized to further improve its performance through the use of techniques such as basic block chaining, elimination of dead code, relaxed condition flags check, and many others. HP’s Dynamo system [2] is a precursor of many of these techniques and we refer the readers to it for a deeper analysis of dynamic compilation techniques. Other available systems that we are aware that employ dynamic compilation techniques include AMD’s SimNow™ [4] and QEMU [6].

To further improve on dynamic compilation techniques, VMs provide a total abstraction of the underlying physical system. A typical VM only interprets kernel mode code, while user mode code is directly executed by the guest machine (note that full virtualization requires the same ISA in the guest and the host). No modification is required in the guest OS or application and they are unaware of the virtualized environment so they execute on the VM just as they would on a physical system. Examples of systems that support full virtualization are VMware [14] and the *kqemu* module of QEMU [5].

Finally, *paravirtualization* is a novel approach to achieving high-performance virtualization on non-virtualizable hardware. In paravirtualization, the guest OS is ported to an idealized hardware layer, which abstracts away all hardware interfaces. Absent upcoming hardware in processor, paravirtualization requires modifications in the guest OS, so that all sensitive operations (such as page table updates or DMA operations) are replaced by explicit calls into the virtualizer API. Xen [3] is currently one of the most advanced paravirtualization layers.

Regarding execution speed, it is clear that interpretation of instructions is the slowest component of functional simulation. Dynamic compilation accelerates interpretation by removing the fetch-decode-translate overhead, but compromises the observability of the system. In other words, in a VM it is much more difficult to extract the instruction-level (or memory-access level) information needed to feed a timing simulator. Interrupting native execution in the code cache to extract statistics is a very expensive operation that requires two context switches and several hundred of cycles of overhead, so it is unfeasible to do so at the granularity of individual instructions.

2.3. Accelerating Both Functional and Timing

We are only aware of few simulation packages that attempt combining fast functional simulation and timing.

PTLsim [23] combines timing simulation with direct host execution to speed up functional simulation in periods in which timing is not activated. During direct execution periods, instructions from the simulated program are executed using native instructions from the host system, rather

than emulating the operation of each instruction. PTLsim does not provide a methodology for fast timing simulation, but simply employs direct execution as a way to skip the initialization part of a benchmark.

PTLsim/X [23] leverages Xen [3] in an attempt to simulate complete systems. The use of paravirtualization allows the simulator to run at the highest privilege level, providing a virtual processor to the target OS. At this level, both the target’s operating system and user level instructions are modeled by the simulator, and it can communicate with Xen to provide I/O when needed by the target OS. PTLsim/X does not however provide a methodology to combine fast timing simulation.

DirectSMARTS [8] combines SMARTS sampling with fast functional simulation. It leverages the *direct execution mode* (emulation mode with binary translation) of RSIM [10] to perform the warming of simulated structures (caches, branch predictor). During emulation, the tool collects a profile of cache accesses and branch outcomes. Before each simulation interval, the collected profile is used to warm-up stateful simulated structures. Although DirectSMARTS is faster than regular SMARTS, it still requires collecting information during functional simulation. This clearly limits further improvements and inhibits the use of more aggressive virtualization techniques.

3. Combining VMs and Timing

In this section we describe the different parts of our simulation environment, as well as the benchmarks and parameters used for calculating results.

3.1. The Functional Simulator

We use AMD’s SimNow™ simulator [4] as the functional simulation component of our system. The SimNow simulator is a fast full-system emulator using dynamic compilation and caching techniques, which supports booting an unmodified OS and execute complex applications over it. The SimNow simulator implements the x86 and x86-64 instruction sets, including system devices and supports unmodified execution of Windows or Linux targets. In full-speed mode, the SimNow simulator’s performance is around 100–200 MIPS (i.e., approximately a 10x slowdown with respect to native execution).

Our extensions enable AMD’s SimNow simulator to switch between full-speed functional mode and sampled-mode. In sampled-mode, AMD’s SimNow simulator produces a stream of events which we can feed to our timing modules to produce the performance estimation. During timing simulation we can also feed timing information back to the SimNow software to affect the application behavior,

a fundamental requirement for full-system modeling. In addition to CPU events, the SimNow simulator also supports generating I/O events for peripherals such as block devices or network interfaces.

In this paper, for the purpose of comparing to other published mechanisms, we have selected a simple test set (uniprocessor, single-threaded SPEC benchmarks), disabled the timing feedback, and limited the interface to generate CPU events (instruction and memory). Although device events and timing feedback would be necessary for complex system applications, they have minimal effect on the benchmark set we use in this paper.

As we described before, the cost of producing these events is significant. In our measurements, it causes a 10x–20x slowdown with respect to full speed, so the use of sampling is mandatory. However, with an appropriate sampling schedule, we can reduce the event-generation overhead to that its effect is minimal to overall simulation time.

3.2. The Timing Simulator

The SimNow simulator’s functional mode subsumes a fixed instruction per cycle (IPC) model. In order to predict the timing behaviour of the complex microarchitecture that we want to model, we have to couple an external timing simulator with AMD’s SimNow software.

For this purpose, in this paper we have adopted PTLsim [23] as our timing simulator. PTLsim is a simulator for microarchitectures of x86 and x86-64 instruction sets, modeling a modern speculative out-of-order superscalar processor core, its cache hierarchy and supporting hardware. As we are only interested in the microarchitecture simulation, we have adopted the classic version of PTLsim (with no SMT/SMP model and no integration with Xen hypervisor [22]) and have disabled its direct execution mode. The resulting version of PTLsim is a normal timing simulator which behaves similarly to existing microarchitecture simulators like SimpleScalar or SMTsim, but with a more precise modeling of the internal x86/x86-64 out-of-order core. We have also modified PTLsim’s front-end to interface directly with the SimNow simulator for the stream of instructions and data memory accesses.

3.3. Simulation Parameters and Benchmarks

Table 1 gives the simulation parameters we use to configure PTLsim. This configuration roughly corresponds to a 3-issue machine with microarchitecture parameters similar to one of the cores of an AMD Opteron™ 280 processor.

In our experiments, we simulate the whole SPEC CPU2000 benchmark suite using the reference input. Benchmarks are simulated until completion or until they reach 240 billion instructions, whatever occurs first. Table 2

Fetch/Issue/Retire Width	3 instructions
Branch Mispred. Penalty	9 processor cycles
Fetch Queue Size	18 instructions
Instruction window size	192 instructions
Load/Store buffer sizes	48 load, 32 store
Functional units	4 int, 2 mem, 4 fp
Branch Prediction	16K-entry gshare; 32K-entry BTB; 16-entry RAS
L1 Instruction Cache	64KB, 2-way, 64B line size
L1 Data Cache	64KB, 2-way, 64B line size
L2 Unified Cache	1MB, 4-way, 128B line size
L2 Unified Cache Hit Lat.	16 processor cycles
L1 Instruction TLB	40 entries, full-associative
L1 Data TLB	40 entries, full-associative
L2 Unified TLB	512 entries, 4-way
TLB pagesize	4KB
Memory Latency	190 processor cycles

Table 1. Timing simulator parameters

shows the reference input used (2nd column) and the number of instructions executed per benchmark (3rd column).

The SimNow simulator guest runs a 64-bit Ubuntu Linux with kernel 2.6.15. The simulation host is a farm of HP ProLiant BL25p server blades with two 2.6GHz AMD Opteron processors running 64-bit Debian Linux. The SPEC benchmarks have been compiled directly in the simulator VM with gcc/g77 version 4.0, with ‘-O3’ optimization level. The simulated execution of the benchmarks is at maximum (guest) OS priority, to minimize the impact of other system processes. The simulation results are deterministic and reproducible. In order to evaluate just the execution of each SPEC benchmark, we restore a snapshot of the VM taken when the machine is idle (except for standard OS housekeeping tasks) and directly invoke the execution of the benchmark from a Linux shell. The timing simulation begins just after the execution command is typed in the OS console.

To simulate SimPoint we interface with AMD’s SimNow software to collect a profile of basic block frequency (Basic Block Vectors [15]). This profile is then used by the SimPoint 3.2 tool [7] to calculate the best simulation points of each SPEC benchmark. Following the indications by Hamerly et al. [9], we have chosen a configuration for SimPoint aimed at reducing accuracy error, while maintaining a high speed: 300 clusters of 1M instructions each. The last column in Table 2 shows the number of simpoints per benchmark as calculated by SimPoint 3.2. Notice how the resulting number of simpoints varies from benchmark to benchmark, depending on the variability of its basic block frequency. For a maximum of 300 clusters, benchmarks have an average of 124.6 simpoints.

For SMARTS, we have used the configuration reported by Wunderlich et al. [21], which assumes that each func-

<i>SPEC benchmark</i>	<i>Ref. input</i>	<i># Instruc. (billions)</i>	<i># SimPoints K=300</i>
gzip	graphic	70	131
vpr	place	93	89
gcc	166.i	29	166
mcf	inp.in	48	86
crafty	crafty.in	141	123
parser	ref.in	240	153
eon	cook	73	110
perlbmk	diffmail	32	181
gap	ref.in	195	120
vortex	lendian1.raw	112	91
bzip2	source	85	113
twolf	ref.in	240	132
wupwise	wupwise.in	240	28
swim	swim.in	226	135
mgrid	mgrid.in	240	124
applu	applu.in	240	128
mesa	mesa.in	240	81
galgel	galgel.in	240	134
art	c756hel.in	56	169
equake	inp.in	112	168
facerec	ref.in	240	147
ammp	ammp-ref.in	240	153
lucas	lucas2.in	240	44
fma3d	fma3d.in	240	104
sixtrack	fort.3	240	235
apsi	apsi.in	240	94

Table 2. Benchmark characteristics

tional warming interval is 97K instructions in length, followed by a detailed warming of 2K instructions, and a full detailed simulation of 1K instructions. This configuration produces the best accuracy results for the SPEC benchmark suite.

For SimPoint and Dynamic Sampling, each simulation interval is preceded by a warming period of 1 million instructions.

4. Dynamic Sampling

In the process of emulating a complete system, a VM performs many different tasks and keeps track of several statistics. These statistics not only serve as a debugging aid for the VM developers, but can also be used as an aid to the emulation itself because they highly correlate with the run-time behavior of the emulated system.

Note that, in the dynamic compilation domain, this property has been observed and exploited before. For example, HP’s Dynamo [2] used its fragment cache (*a.k.a.* code cache or translation cache) hit rate as a metric to detect phase changes in the emulated code. A higher miss rate occurs when the emulated code changes, and Dynamo used this heuristic to force a fragment cache flush. Flushing

whenever this happened proved to be much more efficient than a fine grain management of the code cache employing complex replacement policies.

Our dynamic sampling mechanism stands on similar principles, but with another objective. We are not trying to improve functional simulation or dynamically optimize code, but rather, our goal is to determine representative samples of emulated guest code to speed up timing simulation while maintaining high accuracy.

4.1. Using Virtualization Statistics to Perform Dynamic Sampling

AMD’s SimNow simulator maintains a series of internal statistics collected during the emulation of the system. These statistics measure elements of the emulated system as well as the behavior of its internal structures. The statistics related to the characteristics of the emulated code are similar to those collected by microprocessors hardware counters. For example, the SimNow simulator maintains the number of executed instructions, memory accesses, exceptions, bytes read or written to or from a device. This data is inherent to the emulated software, and at the time is also 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, by running experiments along similar lines of reasoning [12].

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

Among the internal statistics of our functional simulator, in this paper we have chosen three categories in order to show the validity of our dynamic sampling. These categories are the following:

- **Code cache invalidations:** Every time some piece of code is evicted from the translation cache, a counter is incremented. A high number of invalidations in a short period of time indicates a significant change in the code that is being emulated, such as a new program being executed or a major change of phase in the running program.
- **Code exceptions:** Software exceptions, which include system calls, virtual memory page misses and many more are good indicators of a change in the behaviour of the emulated code.

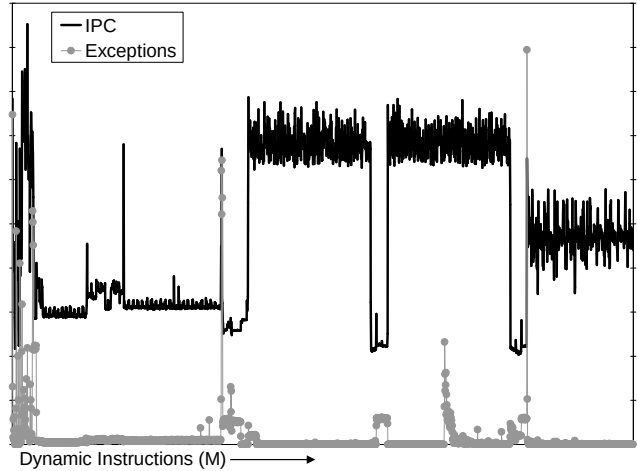


Figure 2. Example of correlation between VM internal statistic and application performance

- **I/O operations:** AMD’s SimNow simulator, like any other system VM, has to emulate the access to all the devices of the virtual environment. This metric detects transfers of data between the CPU and any of the surrounding devices (e.g., disk, video card, or network interface). Usually, applications write data to devices when they have finished a particular task (end of execution phase) and get new data from them at the beginning of a new task (start of a new phase).

Figure 2 shows an example of the correlation that exists between an internal VM statistic and the performance of an application. The graph shows the evolution of the IPC (instructions per cycles) along the execution of the first 2 billion instructions of the benchmark *per lbmk*. Each sample or x-axis point corresponds to 1 million simulated instructions and was collected over a full-timing simulation with our modified PTLsim. The graph also shows the values of one of the internal VM metrics, the number of code exceptions, in the same intervals. We can see that changes in the number of exceptions caused by the emulated code are correlated with changes in the IPC of the application. During the initialization phase (leftmost fraction of the graph) we observe several phase changes, which translate into many peaks in the number of exceptions. Along the execution of the benchmark, every major change in the behavior of the benchmark implies a change in the measured IPC, and also a change in the number of exceptions observed. While VM statistics are not as “fine-grained” as the micro-architectural simulation of the CPU, we believe that they can still be used effectively to dynamically detect changes in the application. We will show later a methodology to use these metrics to perform dynamic sampling.

4.2. Methodology

In order to better characterize Dynamic Sampling, we analyzed the impact that different parameters have to our algorithm, as described in Algorithm 1. The parameters we analyze are the variable to monitor (**var**) and the phase change sensitivity (**S**). The variable to monitor is one of the internal statistics available in the VM. The sensitivity indicates the minimum first-derivative threshold of the monitored variable that triggers a phase change.

Dynamic Sampling employs a third parameter (**max_func**) that allows us to control the generation of timing samples. **max_func** indicates the maximum number of consecutive intervals without timing. When this happens, the algorithm forces a measurement of time in the next interval, which lets assure a minimum number of timing intervals regardless the dynamic behaviour of the sampling.

The control logic of our algorithm inspects the monitored variables at the end of each interval. Whenever the relative change between successive measurements is larger than the sensitivity, it activates full timing simulation for the next interval. During this full timing simulation interval, the VM generates all necessary events for the PTLsim module (which cause it to run significantly slower). At the end of this simulation interval, timing is deactivated, and a new fast functional simulation phase begins. To compute the cu-

mulative IPC, we weight the average IPC of the last timing phase with the duration of the current functional simulation phase, *à la* SimPoint. This process is iterated until the end of the simulation.

4.3. Dynamic Sampling vs. Conventional Sampling

Figure 3 shows an overview of how SMARTS, SimPoint, and Dynamic Sampling determine the simulation samples of an application.

SMARTS (Figure 3.a) employs systematic sampling. It makes use of statistical analysis in order to determine the amount 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 ($\sim 1,000$ instructions) it is crucial for this mechanism to keep micro-architectural structures such as caches and branch predictors warmed-up all the time. For this reason, they perform a functional warming between sampling units. In our environment this means forcing the VM to produce events all the time, preventing it from running at full speed.

The situation is quite similar with SimPoint (Figure 3.b). SimPoint runs a full profile of benchmarks to collect Basic Block Vectors [15] that are later processed using clustering and distance algorithms to determine the simulation points. Figure 3.b shows the IPC distribution of the execution of **Swim** with its reference input. In the figure, different colors visually shade the different phases and we manually associate them with the potential simulation points that SimPoint could decide based on the profile analysis¹. The profiling phase of SimPoint imposes a severe overhead for VMs, since it requires a pre-execution of the complete benchmark. Moreover, as any other kind of profile, its “accuracy” is impacted when input data for the benchmark changes or when it is hard (or impossible) to find a representative training set.

Dynamic Sampling (Figure 3.c) eliminates these drawbacks by determining at emulation time when to sample. We do not require any preprocessing or *a priori* knowledge of the characteristics of the application being simulated. Our sampler monitors some of the internal statistics of the VM, and according to pre-established heuristics, determines when an application is changing to a new phase. When the monitored variable exceeds the sensitivity threshold, the sampler activates the timing simulator for a certain number of instructions, in order to collect a performance measurement of this new phase of the application. The lower the sensitivity threshold, the more the number of timing samples. When the timing sample terminates, the sampler instructs the VM to stop the generation of events and

Algorithm 1: Dynamic Sampling algorithm

```
Data: var = VM statistic to monitor
Data: S = Sensitivity
Data: max_func = Max consecutive functional intervals
Data: num_func = # consecutive functional intervals
Data: timing = Calculate timing?

Set timing = false
/* Main simulation loop */
repeat
  if (timing = false) then
    | Fast functional simulation of this interval
  else
    | Simulate this interval with timing
    | Set timing = false
    | Set num_func = 0
  if ( $\Delta var > S$ ) then
    | Set timing = true
  else
    | Set num_func ++
    if (num_func = max_func) then
      | Set timing = true
    else
      | Set timing = false
until end of simulation
```

¹Although this example represents a real execution, simulation points have been artificially placed to explain SimPoint’s profiling mechanism, but do not come from a real SimPoint profile.

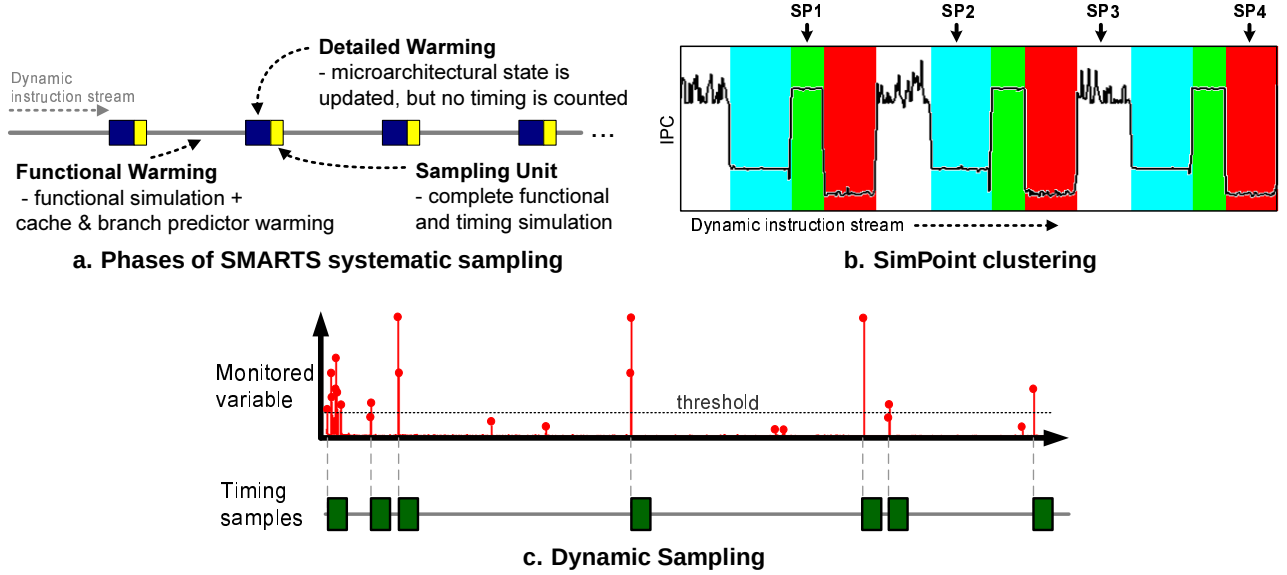


Figure 3. Schemes of SMARTS, SimPoint, and Dynamic Sampling

return to its full speed execution mode until the next phase change is detected.

Unlike SimPoint, we do not need a profile for each input data set, since each simulation determines its own representative samples. We have empirically observed that in many cases our dynamic selection of samples is very similar to what SimPoint statically selects, which improves our confidence of the validity of our choice of monitored VM statistics. We also believe that our mechanism better integrates in a full system simulation setting, while it is going to be much harder for SimPoint to determine the Basic Block Vector distribution of a complete system.

Figure 4 shows an example of the correlation between simulation points as calculated by SimPoint, and simulation points calculated by our Dynamic Sampling. This graph is an extension of the graph shown before in Figure 2, which shows how IPC and number of exceptions change during the execution of benchmark *per lbmk*. Vertical dotted lines indicate six simulation points as calculated by SimPoint 3.2 software from a real profile (labeled SP1, ..., SP6). The graph also shows the six different phases discovered by Dynamic Sampling (stars labeled P1, ..., P6) by using the number of exceptions generated by the emulated software as the internal VM variable to monitor. Note that dynamic discovered phases begins when there is an important change in the monitored variable.

As we can see, there is a strong correlation between the phases detected by SimPoint, and the phases detected dynamically by our mechanism. Dynamic Sampling divides this execution fragment into six phases, which matches SimPoint's selection which also identifies a simulation point from each of these phases ($PN \approx SPN$).

The main difference between SimPoint and Dynamic Sampling is in the selection of the simulation point inside each phase. SimPoint not only determines the program phases, but its offline profiling also allows determining and selecting the most representative interval within a phase. Dynamic Sampling is not able to detect when exactly to start measuring after a phase change, and its only option is to start sampling it right away (i.e., at the beginning of the phase). So, we simply take one sample from the beginning and run functionally until the next phase is detected.

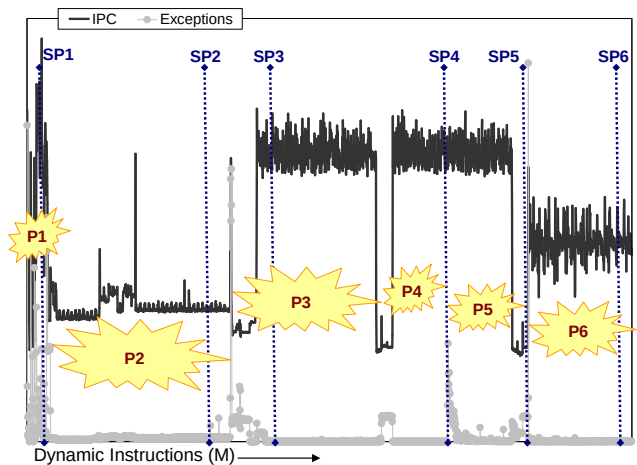


Figure 4. Example of correlation between simulation phases detected by SimPoint and by Dynamic Sampling

5. Results

This section provides simulation results. We first survey our simulation results with a comparison between the accuracy and speed of Dynamic Sampling compared to other mechanisms. Then, we provide an analysis of detailed simulation results of accuracy and speed, as well as results per benchmark.

For Dynamic Sampling, we use the three monitored statistics described in Section 4.1, which will be denoted by CPU (for cache code invalidations), EXC (for code exceptions) and I/O (for I/O operations). Our sampling algorithm uses sensitivity values of 100%, 300% and 500%, interval lengths of 1M, 10M and 100M instructions, and a maximum number of functional intervals of 10 and ∞ (no limit).

5.1. Accuracy vs. Speed Results

Figure 5 shows a summary description of the speed vs. accuracy tradeoffs of the proposed Dynamic Sampling approach and how it compares with conventional sampling techniques. In the x axis we plot the accuracy error vs. what we obtain in a full-timing run (smaller is better). In the logarithmic y axis we plot the simulation execution speedup vs. the full-timing run (larger is better). Each point represents the accuracy error and speed of a given experiment, all relative to a full timing run (speed=1, accuracy error=0). The graph shows four squared points taken as baseline: 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 monitored variable, BB is the sensitivity value, CC is the interval length, and DD is the maximum number of consecutive functional 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.

The point labeled “SMARTS” is a standard SMARTS run, with an error of only 0.5% and a small speedup of 7.4x. Here, we can see how, despite its extraordinary accuracy, SMARTS has to pay the cost of continuous functional warming, as we described before. SMARTS forces AMD’s SimNow simulator to deliver events at every instruction. As we already observed, this slows down the simulator by more than an order of magnitude. The point labeled “SimPoint” is a run of the standard SimPoint with simulation points calculated by off-line profiling (shown in Table 2). With a speedup of 422x, SimPoint is the fastest sam-

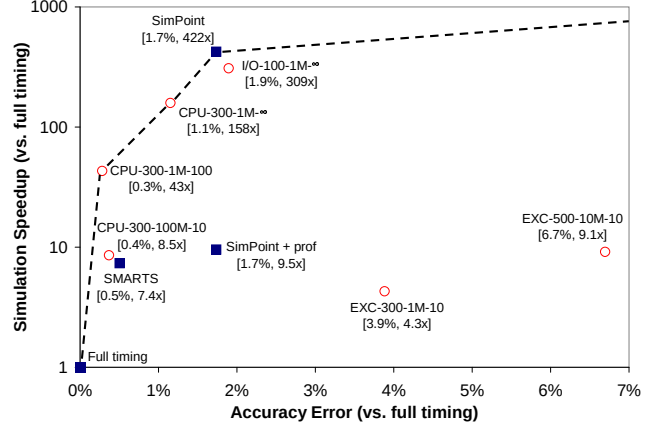


Figure 5. Accuracy vs. Speed results

pling technique. However, as we pointed out previously, SimPoint is really not applicable to system-level simulation because of its need of a separate profiling pass and its impossibility to provide timing feedback. If we also add the overhead of a profiling run (point “SimPoint+prof”), the speed advantage drops at the same level of SMARTS (9.5x).

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 points marked as circles are some of the results of the various Dynamic Sampling experiments. The four points 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 8.5x (similar to SMARTS) to an impressive 309x. An intermediate point with a very good accuracy/speed tradeoff is “CPU-300-1M- ∞ ”, with an accuracy error of 1.1% and a speedup of 158x.

Note, however, that not all Dynamic Sampling heuristics are equally good. For example, points that use EXC as monitored variable are clearly inferior to the rest (and the same is true for other configurations we omitted from the graph for clarity). Hence, it is very important to identify the right variable(s) to monitor and their sensitivity for phase detection: results show that there is a big payoff if we can successfully do so.

5.2. Detailed Accuracy Results

Figure 6 shows the IPC results for our simulated scenarios, averaged over all benchmarks. The first bar represents full timing simulation. The next two bars correspond to SMARTS and SimPoint. The remaining bars show dif-

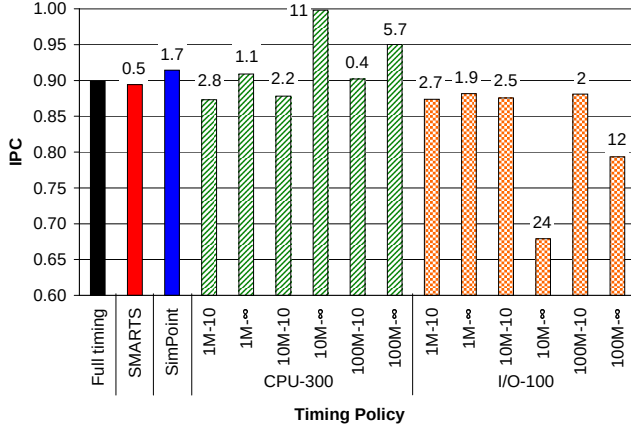


Figure 6. IPC results. Numbers indicate accuracy error (%) over full timing

ferent results of Dynamic Sampling, a first set with CPU as monitored variable and a sensitivity of 300%, and a second set with I/O as variable and 100% as sensitivity. For these sets, we combine interval lengths of 1M, 10M and 100M, with maximum number of functional intervals of 10 and ∞ (no limit). Numbers on top of each bar show the accuracy error (%) compared to the baseline, that is, full timing.

SMARTS provides an IPC error of 0.5% over all benchmarks, while SimPoint provides an IPC error of 1.7%. Dynamic Sampling has a wider range of results. Some configurations, such as CPU-300-100M-10, have as low as 0.4% while others like CPU-300-1M- ∞ go up to 24%. In general, a small interval length of 1M instructions provides good IPC results for almost every monitored variable and sensitivity value. When longer interval lengths are used, it is very important to limit the maximum number of consecutive functional intervals. Using a longer interval implies that small changes in a monitored variable are less noticeable, and so the algorithm activates timing less frequently. We also empirically set a maximum numbers of consecutive functional intervals ($max_func = 10$), to ensure that a minimum number of measurement points is always taken. This provides a better timing characterization of the benchmark, translating into a much higher accuracy.

Figure 8 shows IPC results per individual benchmarks. Results are provided for full timing, SMARTS, SimPoint, and Dynamic Sampling with CPU-300-1M- ∞ . As shown before in Figure 5, this configuration provides very good results for both accuracy and speed.

Overall, SMARTS provides the best accuracy results for 16 out of the 26 SPEC CPU200 benchmarks, with an accuracy error of only 0.1% in *mcf* or 0.22% in *wupwise*. On the contrary, it provides the worst results for *crafty*, with an accuracy error of 8%. SimPoint provides the best

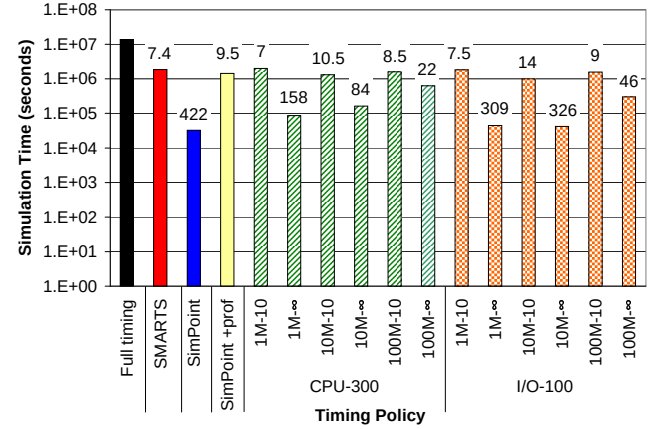


Figure 7. Simulation time results (y axis is logarithmic). Numbers indicate speedup over full timing

accuracy results for 9 out of the 26 benchmarks, with an accuracy error of only 0.37% in *perlbnk* and 0.48% in *gcc*. However, SimPoint is the worst technique for *gap* and *ammp*, with accuracy errors over 20%.

Dynamic Sampling provides the best accuracy results only in two benchmarks, *vpr* (0.36%) and *crafty* (0.9%). However, the results for the rest of benchmarks are quite consistent, and only exceed the 10% boundary for *applu* and *art*.

5.3. Detailed Speed Results

Figure 7 shows the simulation time (in seconds) of the different simulated configurations. Numbers shown over the bars indicate the speedup over the baseline (full timing).

As expected, SMARTS speedup is rather limited. The need for continuous functional sampling constrains its potential in VM environments. SimPoint, on the other hand, provides very fast simulation time. On average, simulations with SimPoint execute around 7% of the total instructions of the benchmark, which translates in an impressive 422x speedup.

However, SimPoint simulation time does not account for the time required to calculate the profile of basic blocks and the execution of the SimPoint 3.2 tool itself. The fourth bar in Figure 7 shows the complete simulation time to perform a SimPoint simulation (including determination of Basic Block Vectors and calculation of simulation points and weights). The need for SimPoint to perform a full simulation of the benchmark requires the VM to generate events, and limits its potential speed. The total simulation time of SimPoint increases by two orders of magnitude.

Finally, Figure 7 also shows simulation time of Dy-

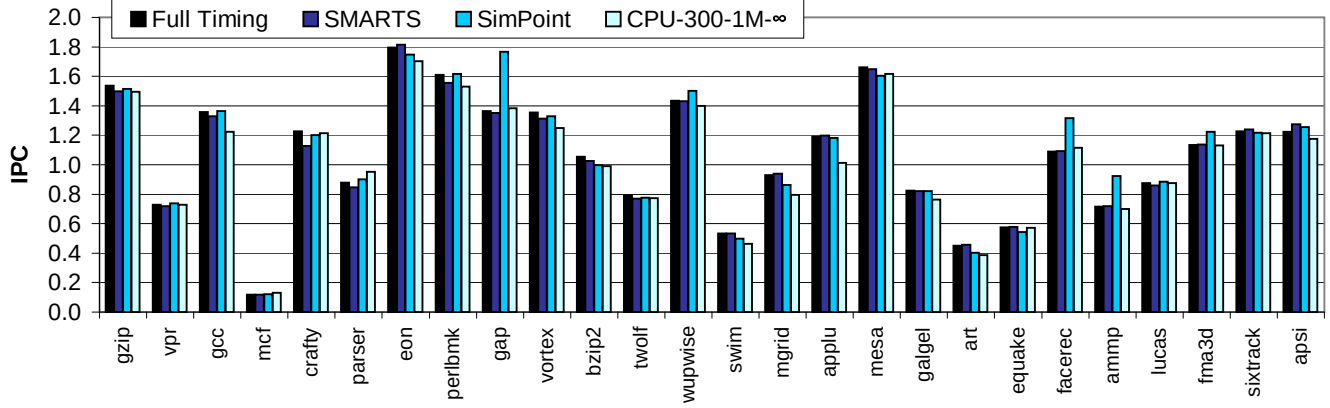


Figure 8. IPC results per benchmark

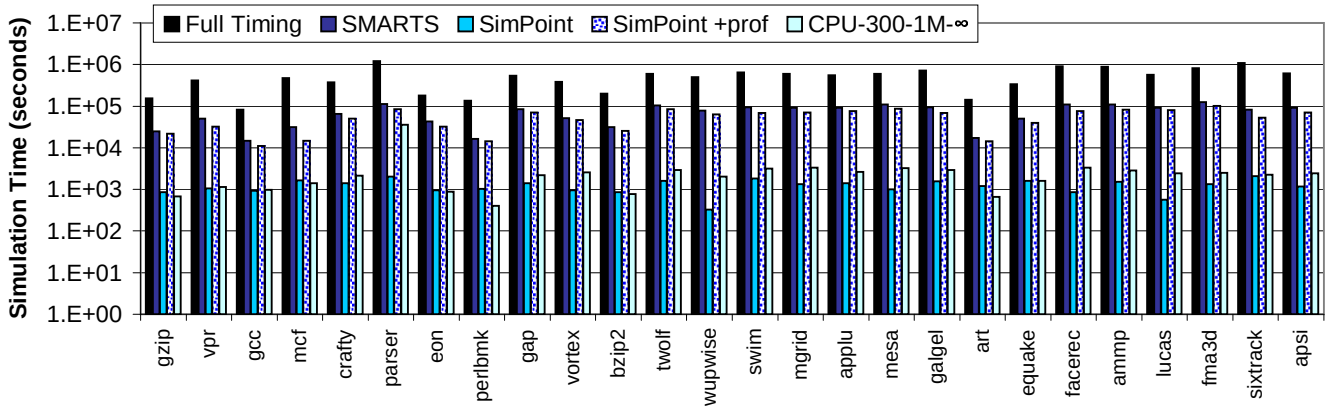


Figure 9. Simulation time per benchmark (y axis is logarithmic)

namic Sampling. The best speedup results are obtained with small intervals and no limits to functional simulation ($max_func = \infty$). On the contrary, larger intervals and limits to functional simulation lengths cause simulation speed to decrease at the same level of SMARTS and SimPoint+prof. Our best configurations are able to provide a simulation speed similar to that provided by SimPoint, without requiring any previous static analysis.

Figure 9 provides simulation time per benchmark. On average, a SPEC CPU2000 benchmark with a single *ref* input takes 6 days to be simulated with full timing in our simulation environment, with a maximum of 14 days for *parser* and a minimum of 23 hours for *gcc*. SMARTS reduces simulation time required by SPEC CPU2000 to an average of 20 hours per benchmark. SimPoint further reduces simulation time to only 21 minutes per benchmark on average. Simulation time in SimPoint is directly proportional to the number of simulation points established per benchmark. For example, *wupwise* only has 28 simpoints, and hence gets simulated in 5.5 minutes, while *sixtrack* has 235 simpoints and gets simulated in 35 minutes.

The simulation time of Dynamic Sampling also depends on the particular benchmark, since the sampling selection varies according to the different phases dynamically detected. Overall, the simulation time of Dynamic Sampling is equivalent to that obtained with SimPoint without considering its profiling time (except for few benchmarks — *parser*, *wupwise*, *facerec*, *lucas*—), and clearly better than SMARTS and Simpoint+prof for every benchmark. Thus, with Dynamic Sampling, *perlbnk* is simulated in 6.7 minutes (with a 4.9% of accuracy error), while *parser* takes 9.8 hours (with a 7.4% of accuracy error).

6. Conclusions

We believe that our approach points to a promising direction for next-generation simulators. In the upcoming era of multiple cores and ubiquitous parallelism, we have to upgrade our tools and methodology so that they may be applied to a complex system environment where the CPU is nothing more than a component. In a complex system, being able to characterize the full computing environments,

including OS and system tasks, in the presence of variable parameters and with a reasonable accuracy is becoming a major challenge in the industry. In this world, it is hard to see the applicability of techniques like SimPoint, which reach excellent accuracy but rely on a full profiling pass on repeatable inputs.

What we propose is novel in several ways: to the best of our knowledge we are the first to advocate a system that combines fast VMs and accurate architectural timing. Our approach enables modeling a complete system including peripherals running full, unmodified operating system and real applications with unmatched execution speed. At the same time, we can support a timing accuracy that approximates the best existing sampling mechanisms.

The Dynamic Sampling techniques that we propose in this paper represent a first step in the direction of developing a full-system simulator for “modern” computing systems. They combine the outstanding speed and functional completeness of fast emulators with the high accuracy of sampled timing models. We have shown that, depending on the chosen heuristics, it is possible to find simulation configurations that excel in accuracy (8.5x speed and 0.4% error vs. full timing simulation), or even more interestingly, in speed (309x speedup and 1.9% error). At the same time, our approach is fully dynamic, does not require any *a priori* profiling pass, and provides timing feedback to the functional simulation. This puts us one step closer to being able to faithfully simulate a complete multi-core, multi-socket system, and we believe represents a major advancement in the area of computer architecture simulation.

Acknowledgments

We especially thank AMD’s SimNow team for helping us and providing the necessary infrastructure to perform the experiments presented in this paper.

References

- [1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, Feb. 2002.
- [2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Procs. of the 2000 Conf. on Programming Language, Design and Implementation*, pages 1–12, June 2000.
- [3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Procs. of the 19th Symp. on Operating Systems Principles*, pages 164–177, Oct. 2003.
- [4] B. Barnes and J. Slice. SimNow: A fast and functionally accurate AMD X86-64 system simulator. Tutorial at *2005 Intl. Symp. on Workload Characterization*, Oct. 2005.
- [5] F. Bellard. QEMU webpage. <http://www.qemu.org>.
- [6] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX 2005 Annual Technical Conf., FREENIX Track*, pages 41–46, Apr. 2005.
- [7] B. Calder. SimPoint webpage. <http://www.cse.ucsd.edu/~calder/simpoint>.
- [8] S. Chen. Direct SMARTS, accelerating microarchitectural simulation through direct execution. Master’s thesis, Electrical & Computer Engineering, Carnegie Mellon University, June 2004.
- [9] G. Hamerly, E. Perelman, J. Lau, B. Calder, and T. Sherwood. Using machine learning to guide architecture simulation. *Journal of Machine Learning Research*, 7:343–378, Feb. 2006.
- [10] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. Rsim: Simulating shared-memory multiprocessors with ILP processors. *Computer*, 35(2):40–49, Feb. 2002.
- [11] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. *Workload Character. 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 *Procs. of the Intl. Symp. on Performance Analysis of Systems and Software*, pages 236–247, Mar. 2005.
- [13] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hällberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, Feb. 2002.
- [14] M. Rosenblum. VMWare. <http://www.vmware.com>.
- [15] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Procs. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, Oct. 2002.
- [16] J. E. Smith and R. Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, May 2005.
- [17] D. M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conf.*, pages 819–828, Dec. 1996.
- [18] M. Van Biesbrouck, L. Eeckhout, and B. Calder. Efficient sampling startup for sampled processor simulation. In *Procs. of the Intl. Conf. on High Performance Embedded Architectures & Compilers*, Nov. 2005.
- [19] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. TurboSMARTS: Accurate microarchitecture simulation sampling in minutes. *SIGMETRICS Perform. Eval. Rev.*, 33(1):408–409, June 2005.
- [20] Wikipedia. Comparison of virtual machines. http://en.wikipedia.org/wiki/Comparison_of_virtual_machines.
- [21] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Procs. of the 30th Annual Intl. Symp. on Computer Architecture*, pages 84–97, June 2003.
- [22] M. T. Yourst. PTLsim user’s guide and reference. <http://www.ptlsim.org>.
- [23] M. T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Procs. of the Intl. Symp. on Performance Analysis of Systems and Software*, Apr. 2007.