Full Speed Ahead: Detailed Architectural Simulation at Near-Native Speed

Andreas Sandberg Erik Hagersten David Black-Schaffer

***Abstract*—Popular microarchitecture simulators are typically several orders of magnitude slower than the systems they simulate. This leads to two problems: First, due to the slow simulation rate, simulation studies are usually limited to the first few billion instructions, which corresponds to less than 10% the execution time of many standard benchmarks. Since such studies only cover a small fraction of the applications, they run the risk of reporting unrepresentative application behavior unless sampling strategies are employed. Second, the high overhead of traditional simulators make them unsuitable for hardware/software co-design studies where rapid turn-around is required.**

In spite of previous efforts to parallelize simulators, most commonly used full-system simulations remain single threaded. In this paper, we explore a simple and effective way to parallelize sampling full-system simulators. In order to simulate at high speed, we need to be able to efficiently fast-forward between sample points. We demonstrate how hardware virtualization can be used to implement highly efficient fast-forwarding in the standard gem5 simulator and how this enables efficient execution between sample points. This extremely rapid fast-forwarding enables us to reach new sample points much quicker than a single sample can be simulated. Together with efficient copying of simulator state, this enables parallel execution of sample simulation. These techniques allow us to implement a highly scalable sampling simulator that exploits sample-level parallelism.

We demonstrate how virtualization can be used to fast-forward simulators at 90% of native execution speed on average. Using virtualized fast-forwarding, we demonstrate a parallel sampling simulator that can be used to accurately estimate the IPC of standard workloads with an average error of 2.2% while still reaching an execution rate of 2.0 GIPS (63% of native) on average. We demonstrate that our parallelization strategy scales almost linearly and simulates one core at up to 93% of its native execution rate, 19 000x faster than detailed simulation, while using 8 cores.

I. Introduction

Simulation is commonly used to evaluate new proposals in computer architecture and to understand complex hardware/software interactions. However, traditional simulation is very slow. While the performance of computer systems have steadily increased, simulators have become increasingly complex, and their performance relative to the simulated systems has decreased. A typical full-system simulator for a single out-of-order (OoO) processor executes around 0.1 million instructions per second (MIPS) on a modern processor that peaks at several billion instructions per second per core. Even fast, simplified, simulation modes typically execute at only 1–10 MIPS. The slow simulation rate is a severe limitation when evaluating new high-performance computer architectures or researching hardware/software interactions. There is therefore a need for efficient sampling simulators that are able to fast-forward simulation at near-native speed.

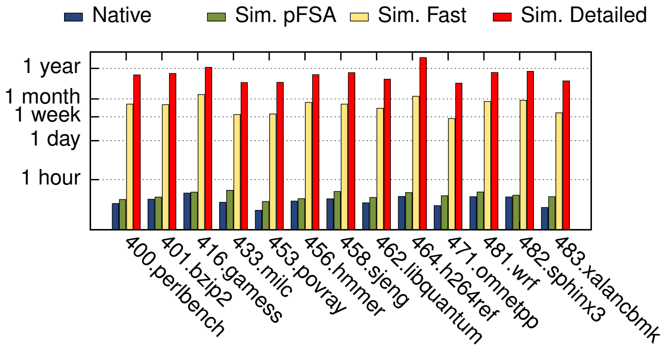


Figure 1: Native, our parallel sampler (pFSA), and projected execution times using gem5's functional and detailed out-of-order CPUs for a selection of SPEC CPU2006 benchmarks.

Many common benchmarks take an exorbitant amount of time to simulate in detail to completion. This is illustrated by Figure 1 which compares execution times of native execution, our parallel sampling method (pFSA), and project simulation times using the popular gem5[1] full-system simulator. The low simulation speed has several undesirable consequences: 1) In order to simulate interesting parts of a benchmark, researchers often fast-forward to a point of interest (POI). In this case, fast forwarding to a new a simulation point close to the end of a benchmark takes between a week and a month, which makes this approach painful or even impractical. 2) Since fast-forwarding is relatively slow and a sampling simulator can never execute faster than the fastest simulation mode, it is often impractical to get good full-application performance estimates using sampling techniques. 3) Interactive use is slow and painful. For example, setting up and debugging a new experiment would be much easier if the simulator could execute at more human-usable speeds.

Many researchers have worked on improving simulator performance. One popular strategy has been to sample execution. The SMARTS[2] methodology uses periodic sampling, wherein the simulator runs in a fast mode most of the time and switches to a detailed simulation mode to measure performance. A similar idea, SimPoint[3], uses stored checkpoints of multiple samples that represent the dominant parts of an application.

In this work, we propose a simple but effective parallel sampling methodology that uses *hardware virtualization* to *fast-forward* between samples at near-native speed and parallelization to overlap detailed simulation and fast-forwarding. We demonstrate an implementation of our sampling methodology for the popular gem5 full-system simulation environment. However, the methodology itself is general and can be applied to other simulation environments. In our experiments, we show that our implementation scales almost linearly to close to native speed of execution resulting in a peak performance in excess of 4 GIPS.

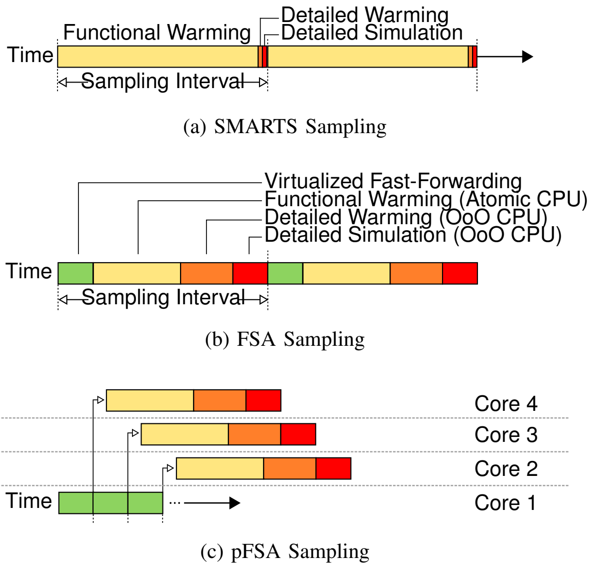
To accomplish this, we extend gem5 with a new CPU module that uses the hardware virtualization support available in current ARM- and x86-based hardware to execute directly on the physical host CPU. Our *virtual CPU module* uses standard Linux interfaces, such as the Linux Kernel-based Virtual Machine[4] (KVM) that exposes hardware virtualization to user space. This virtual CPU module is similar to that of PTLsim[5], but differs on two crucial points, both of which stem from PTLsim's use of the Xen para-virtualization environment. Since PTLsim depends on Xen, it presents a para-virtualized environment to the simulated system. This means that the simulated system needs to be aware of the Xen environment to function correctly and it does not simulate many important low-level hardware components, such as interrupt timers or IO devices. In addition, the use of Xen makes it difficult to use PTLsim in a shared environments (e.g., a shared cluster), which is not the case for our KVM-based implementation. Since KVM is provided as a standard component in Linux, we have successfully used our CPU module on shared clusters without modifying the host's operating system.

Having support for virtualization in gem5 enables us to implement extremely efficient *Virtual Fast-Forwarding* (VFF), which executes instructions at close to native speed. By itself, VFF overcomes some of the limitations of traditional simulation environments. Using VFF, we can quickly execute to a POI anywhere in a large application and then switch to a different CPU module for detailed simulation, or take a checkpoint for later use. Due to the its speed, it is feasible to work interactively with the simulator while debugging and setting up the simulation environment.

VFF enables us to rapidly fast-forward the simulator at near-native speed. We use this capability to implement a highly efficient sampling simulator. We demonstrate how this simulator can be parallelized using standard operating system techniques, which enables us to overlap sample simulation with fast-forwarding. We call this parallel simulator pFSA for Parallel Full Speed Ahead. Similar parallelization techniques[6], [7] have previously been applied to the Pin[8] dynamic instrumentation engine in order to hide instrumentation costs. However, unlike Pin-based approaches, we are not limited to user-space profiling.

Our contributions are:

- We present a new *virtual CPU module* in gem5 which uses standard Linux interfaces and executes code at near-native speed. We are actively working on contributing this to the rest of the gem5 community.
- We demonstrate how hardware virtualization can be used to implement *fast and accurate simulation sampling* (2.0% IPC error on average).
- We present a simple strategy to parallelize sampling simulators which results in almost linear speedup to close to native speed (63% of native execution, 2.0 GIPS on average) for a selection of SPEC CPU2006 benchmarks.
- We present a method that estimates the accuracy loss due to cache warming inaccuracies, and demonstrate how it can be integrated into the sampling framework with low overhead (adding 3.9% overhead on average).



(a) SMARTS Sampling

Virtualized Fast-Forwarding
Functional Warming (Atomic CPU)
Detailed Warming (OoO CPU)
Detailed Simulation (OoO CPU)
Time
Sampling Interval

(b) FSA Sampling

(c) pFSA Sampling

Figure 2: Comparison of how different sampling strategies interleave different simulation modes.

II. Overview of FSA Sampling

Until now, detailed simulation has been painfully slow. To make simulators usable for larger applications, many researchers[2], [9], [10], [11], [3] have proposed methods to sample the simulation. With sampling, the simulator can run in a faster, less detailed mode between samples, and only spend time on slower detailed simulation for the individual samples. Design parameters such as sampling frequency, cache warming strategy, and fast forwarding method give the user the ability to control the trade-off between performance and accuracy to meet his or her needs.

SMARTS[2] is a well-known sampling methodology which uses three different modes of execution to balance accuracy and simulation overhead. The first mode, *functional warming*, is the fastest mode and executes instructions without simulating timing, but still simulates caches and branch predictors to maintain long-lasting microarchitectural state. This mode moves the simulator from one sample point to another and executes the bulk of the instructions. The second mode, *detailed warming*, simulates the entire system in detail using an OoO CPU model and warms the CPU's internal structures (e.g., load and store buffers). The third mode, *detailed sampling*, simulates the system in detail and takes the desired measurements. The interleaving of these sampling modes is shown in Figure 2a.

SMARTS uses a technique known as *always-on* cache and branch predictor warming, which guarantees that these resources are warm when a sample is taken. This makes it trivial to ensure that the long-lived microarchitectural state (e.g., caches and branch predictors) is warm. However, the overhead of always-on cache warming, which effectively prevents efficient native execution, is significant. We trade-off the guarantees provided by always-on cache and branch predictor warming for dramatic performance improvements (in the order of 1000x) and demonstrate a technique that can be used to detect and estimate warming errors.

In traditional SMARTS-like sampling, the vast majority of the simulation time is spent in the functional warming mode[2], [9] as it executes the vast majority of the instructions. To reduce the overhead of this mode, we use VFF to execute instructions at near-native speed on the host CPU when the simulator is executing between samples. However, we cannot directly replace the functional warming mode with native execution using VFF, as VFF can not warm the simulated caches and branch predictors. Instead, we add it as a new execution mode, *virtualized fast-forward*, which uses VFF to execute between samples. After executing to the next sample at near-native speed in the virtualized fast-forward mode, we switch to the functional warming mode, which now only needs to run long enough to warm caches and branch predictors. This allows us to execute the vast majority of our instructions at near native speed through hardware virtualization (Figure 2b). We call this sampling approach Full Speed Ahead (FSA) sampling. With this approach, the majority of the instructions (typically more than 95%) now execute in the (enormously faster) virtualized fast-forward mode instead of the simulated functional warming mode.

Despite executing the majority of the *instructions* natively, FSA still spends the majority of its *time* in the non-virtualized simulation modes (typically 75%–95%) to warm and measure sample points. To parallelize this sample simulation we need to do two things: copy the simulator state for each sample point (to allow them to execute independently), and advance the simulator to the next simulation point before the previous ones have finished simulating (to generate parallel work). We implement such a parallel simulator by continuously running the simulator in the virtualized fast-forward mode, and cloning the simulator state when we want to take a sample. We then do a detailed simulation of the cloned sample in parallel with the continued fast-forwarding of the original execution. If we can copy the system state with a sufficiently low overhead, the simulator will scale well, and can reach near-native speeds. We call this simulation mode Parallel Full Speed Ahead (pFSA) sampling. pFSA has the same execution modes as FSA, but unlike FSA the functional and detailed modes execute in parallel with the virtualized fast-forward mode (Figure 2c).

Since FSA and pFSA use limited warming of caches and branch predictors, there is a risk of insufficient warming which can lead to incorrect simulation results. To detect and estimate the impact of limited warming, we devise a simulation strategy that allows us to run the detailed simulation for both the optimistic (sufficient warming) and pessimistic (insufficient warming) cases. We use our efficient state copying mechanism to quickly re-run detailed warming and simulation without re-running functional warming. This results in a very small overhead since the simulator typically spends less than 10% of its execution time in the detailed modes. The difference between the pessimistic and optimistic cases gives us insight into the impact of functional warming.

III. Background

A. gem5: Full-System Discrete Event Simulation

Full-system simulators are important tools in computer architecture research as they allow architects to model the performance impact of new features on the whole computer system including the operating system. To accurately simulate the behavior of a system, they must simulate all important components in the system, including CPUs, the memory system, and the I/O and the storage. In most simulators the components are designed as modules, enabling users to plug in new components relatively easily.

Simulators based on discrete event simulation handle time by maintaining a queue of events that happen at specific times. Each event is associated with an event handler that is executed when the event is triggered. For example, an event handler might simulate one clock cycle in a CPU. New events are normally only scheduled (inserted into the queue) by event handlers. The main loop in a discrete event simulator takes the first event from the queue and executes its event handler. It continues to do so until it encounters an exit event or the queue is empty. As a consequence of executing discrete events from a queue, the time in the simulated system progresses in discrete steps of varying length, depending on the time between events in the queue.

gem5[1] is a discrete event full-system simulator, which provides modules for most components in a modern system. The standard gem5 distribution includes several CPU modules, notably a detailed superscalar *OoO CPU module* and a simplified faster *functional CPU module* that can be used to increase simulation speed at a loss of detail. The simulated CPU modules support common instruction sets such as ARM, SPARC, and x86. Due to the design of the simulator, all of the instruction sets use the same pipeline models. In addition, gem5 includes memory system modules (GEMS[12] or simplified MOESI), as well a DRAM module, and support for common peripherals such as disk controllers, network interfaces, and frame buffers.

In this paper, we extend gem5 to add support for hardware virtualization through a new *virtual CPU module* and leverage the speed of this new module to add support for parallel sampling. The virtual CPU module can be used as a drop-in replacement for other CPU modules in gem5, thereby enabling rapid execution. Since the module supports the same gem5 interfaces as simulated gem5 CPU modules, it can be used for checkpointing and CPU module switching during simulation.

B. Hardware Virtualization

Virtualization solutions have traditionally been employed to run multiple operating system instances on the same hardware. A layer of software, a *virtual machine monitor* or *VMM*, is used to protect the different operating systems from each other and provide a virtual view of the system. The VMM protects the virtual machines from each other by intercepting instructions that are potentially dangerous, such as IO or privileged instructions. Dangerous instructions are then simulated to give the software running in the virtual machine the illusion of running in isolation on a real machine[1]. Early x86 virtualization solutions (e.g., VMware) used binary rewriting of privileged code to intercept dangerous instructions and complex logic to handle the mapping between addresses in the guest and host system. As virtualization gained popularity, manufacturers started adding hardware virtualization extensions to their processors. These extensions allow the VMM to intercept dangerous instructions without binary rewriting and provide support for multiple layers of address translation (directly translating from guest virtual addresses to host physical addresses). Since these extensions allow most of the code in a virtual machine to execute natively, many workloads execute at native speed.

The goals of virtualization software and traditional computer architecture simulators are very different. One of the major differences is how device models (e.g, disk controllers) are implemented. Traditional virtualization solutions typically prioritize performance, while architecture simulators focus on accurate timing and detailed hardware statistics. Timing sensitive components in virtual machines typically follow the real-time clock in the host, which means that they follow wall-clock time rather than a simulated time base. Integrating support for hardware virtualization into a simulator such as gem5 requires us to ensure that the virtual machine and the simulator have a consistent view of devices, time, memory, and CPU state. We describe these implementation details in Section IV-A.

IV. Implementation

In order to implement a fast sampling simulator, we need to support extremely fast fast-forwarding as most instructions will be executed in the fast-forward mode. We implement rapid fast-forwarding using hardware virtualization which executes code natively. To further improve the performance of the simulator, we overlap fast-forwarding and sample simulation by executing them in parallel. This requires efficient cloning of the simulator's internal state, which we implement using copy-on-write techniques. While our implementation is gem5-specific, we believe that the techniques used are portable to other simulation environment.

A. Hardware Virtualization in gem5

Our goal is to accelerate simulation by off-loading some instructions executed in the simulated system to the hardware CPU. This is accomplished by our virtual CPU module using hardware virtualization extensions to execute code natively at near-native speed. We designed the virtual CPU module to allow it to work as a drop-in replacement for the other CPU modules in gem5 (e.g., the OoO CPU module) and to only require standard features in Linux. This means that it supports gem5 features like CPU module switching during simulation and runs on off-the-shelf Linux distributions.

Integrating hardware virtualization in a discrete event simulator requires that we ensure consistent handling of 1) simulated devices, 2) time, 3) memory, and 4) processor state. First, simulators and traditional virtualization environments both need to provide a device model to make software believe it is running on a real system. We interface the virtual CPU with gem5's device models (e.g., disk controllers, displays, etc.), which allows the virtual CPU to use the same devices as the simulated CPUs. Second, discrete event simulators and traditional virtualization environments handle time in fundamentally different ways. For example, a virtualization environment uses real, wall-clock, time to schedule timer interrupts, whereas a discrete event simulator uses a simulated time base. Interfacing the two requires careful management of the time spent executing in the virtual CPU. Third, full-system simulators and virtualization have different memory system requirements. Most simulators assume that processor modules access memory through a simulated memory system, while the virtual CPU requires direct access to memory. In order to execute correctly, we need to make sure that simulated CPUs and virtual CPUs have a consistent view of memory. Fourth, the state of a simulated CPU is not directly compatible with the real hardware, which makes it hard to transfer state between a virtual CPU and a simulated CPU. These issues are discussed in detail below:

Consistent Devices: The virtualization layer does not provide any device models. A CPU normally communicates with devices through memory mapped IO and devices request service from the CPU through interrupts. Memory accesses to IO devices (and IO instructions such as `in` and `out`) are intercepted by the virtualization layer, which stops the virtual CPU and hands over control to gem5. In gem5, we synthesize a memory access that is inserted into the simulated memory system, allowing the access to be seen and handled by gem5's device models. IO instructions are treated like normal memory-mapped device accesses, but are mapped to a special address range in gem5's simulated memory system. When the CPU model sees an interrupt from a device, it injects it into the virtual CPU using KVM's interrupt interface.

Consistent Time: Simulating time is difficult because device models (e.g., timers) execute in simulated time, while the virtual CPU executes in real time. A traditional virtualization environment solves this issue by running device models in real time as well. For example, if a timer is configured to raise an interrupt every second, it would setup a timer on the host system that fires every second and injects an interrupt into the virtual CPU. In a simulator, the timer model inserts an event in the event queue one second into the future. The simulator then executes instructions, cycle by cycle, until it reaches the timer event. At this point, the timer model raises an interrupt in the CPU. To make device models work reliably, we need to bridge this gap between simulated time and the time as perceived by the virtual CPU.

We address the difference in timing requirements between the virtual CPU and the gem5 device models by restricting the amount of time the virtual CPU is allowed to execute between simulator events. When the virtual CPU is started, it is allowed to execute until a simulated device requires service (e.g., raises an interrupt or starts a delayed IO transfer). This is accomplished by looking into the event queue before handing

[1] This is not strictly true, the virtualization software usually exposes virtual devices that provide more efficient interfaces than simulated hardware devices.

over control to the virtual CPU. If there are events scheduled, we use the time until the next event to determine how long the virtual CPU should execute before handling the event. Knowing this, we schedule a timer that interrupts the virtual CPU at the correct time to return control to the simulator, which handles the event.

Due to the different execution rates between the simulated CPU and the host CPU (e.g., a server simulating an embedded system), we need to scale the host time to make asynchronous events, such as interrupts, happen with the right frequency relative to the executed instructions. For example, when simulating a CPU that is slower than the host CPU, we scale time with a factor that is less than one (i.e., we make device time faster relative to the CPU). This makes the host CPU seem slower as timer interrupts happen more frequently relative to the instruction stream. Our current implementation uses a constant conversion factor, but future implementations could determine this value automatically using sampled timing-data from the OoO CPU module.

Consistent Memory: Interfacing between the simulated memory system and the virtualization layer is necessary to transfer state between the virtual CPU module and the simulated CPU modules. First, the virtual machine needs to know where physical memory is located in the simulated system and where it is allocated in the simulator. Since gem5 stores the simulated system's memory as contiguous blocks of physical memory, we can look at the simulator's internal mappings and install the same mappings in the virtual system. This gives the virtual machine and the simulated CPUs the same view of memory. Second, since virtual CPUs do not use the simulated memory system, we need to make sure that simulated caches are disabled when switching to the virtual CPU module. This means that we need to write back and invalidate all simulated caches when switching to the virtual CPU. Third, accesses to memory-mapped IO devices need to be simulated. Since IO accesses are trapped by the virtualization layer, we can translate them into simulated accesses that are inserted into the simulated system to access gem5's simulated devices.

Consistent State: Converting between the processor state representation used by the simulator and the virtualization layer, requires detailed understanding of the simulator internals. There are several reasons why a simulator might be storing processor state in a different way than the actual hardware. For example, in gem5, the x86 flag register is split across several internal registers to allow more efficient dependency tracking in the OoO pipeline model. Another example are the registers in the x87 FPU: the real x87 stores 80-bit floating point values in its registers, while the simulated x87 only stores 64-bit values. Similar difficulties exist in the other direction. For example, only one of the two interfaces used to synchronize FPU state with the kernel updates the SIMD control register correctly. We have implemented state conversion to give gem5 access to the processor state using the same APIs as the simulated CPU modules. This enables online switching between virtual and simulated CPU modules as well as simulator checkpointing and restarting.

Since our virtual CPU module integrates seamlessly with the rest of gem5, we can use it transfer state to and from other simulated CPU modules. This allows the virtual CPU module to be used as a plug-in replacement for the existing CPU modules whenever simulation accuracy can be traded off for execution speed. For example, it can be used to implement efficient performance sampling by fast-forwarding to points of interest far into an application, or interactive debugging during the setup phase of an experiment.

B. Cloning Simulation State in gem5

Exposing the parallelism available in a sampling simulator requires us to be able to overlap the detailed simulation of multiple samples. When taking a new sample, the simulator needs to be able to start a new worker task (process or thread) that executes the detailed simulation using a copy of the simulator state at the time the sample was taken. Copying the state to the worker can be challenging since the state of the system (registers and RAM) can be large. There are methods to limit the amount of state the worker[9] needs to copy, but these can complicate the handling of miss-speculation. We chose to leverage the host operating system's copy-on-write (CoW) functionality to provide each sample with its own copy of the full system state.

In order to use the CoW functionality in the operating system, we create a copy of the simulator using the fork system call in UNIX whenever we need to simulate a new sample. The semantics of fork gives the new process (the child) a lazy copy (via CoW) of most of the parent process's resources. However, when forking the FSA simulator, we need to solve two problems: shared file handles between the parent and child and the inability of the child to use the same KVM virtual machine that the parent is using for fast-forwarding. The first issue simply requires that the child reopens any file it intends to use. To address the child's inability to use the parent's KVM virtual machine, we need to immediately switch the child to a non-virtualized CPU module upon forking. Since the virtual CPU module used for fast-forwarding can be in an inconsistent state (e.g., when handling IO or delivering interrupts), we need to prepare for the switch in the parent before calling fork (this is known as draining in gem5). By preparing to exit from the virtualized CPU module before forking, we allow the child process to switch to a simulated CPU without having to execute in the virtualized (KVM) CPU module.

One potential problem when using fork to copy the simulation state is that the parent and child will use the same system disk images. Writes from one of the processes could easily affect the other. To avoid this we configure gem5 to use copy-on-write semantics and store the disk writes in RAM.

Our first implementation of the parallel simulator suffered from disappointing scalability. The primary reason for this poor scaling was due to a large number of page faults, and a correspondingly large amount of time spent in the host kernel's page fault handler. These page faults occur as a result of the operating system copying data on writes to uphold the CoW semantics which ensure that the child's copy of the simulated system state is not changed by the fast-forwarding parent. An interesting observation is that most of the cost of copying a page is in the overhead of simply taking the page fault; the actual copying of data is comparatively cheap. If the executing code exhibits decent spatial locality, we would therefore expect to dramatically reduce the number of page faults and their overhead by increasing the page size. In practice, we experienced much better performance with huge pages enabled.

C. *Warming Error Estimation*

We estimate the errors caused by limited warming by re-running detailed warming and simulation without re-running functional warming. We implement this by cloning the warm simulator state (forking) before entering the detailed warming mode. The new child then simulates the pessimistic case (insufficient warming), meanwhile the parent waits for the child to complete. Once the child completes, the parent continues to execute and simulates the optimistic case (sufficient warming).

We currently only support error estimation for caches (we plan to extend this functionality to TLBs and branch predictors), where the optimistic and pessimistic cases differ in the way we treat *warming misses*, i.e. misses that occur in sets that have not been fully warmed. In the optimistic case, we assume all warming misses are actual misses (i.e., sufficient warming). This may underestimate the performance of the simulated cache as some of the misses might have been hits had the cache been fully warmed. In the pessimistic case, we assume that warming misses are hits (i.e., worst-case for insufficient warming). This overestimates the performance of the simulated cache since some of the hits might have been capacity misses.

V. Evaluation

To evaluate our simulator we investigate three key characteristics: functional correctness, accuracy of sampled simulation, and performance. To demonstrate that the virtual CPU module integrates correctly with gem5, we perform two experiments that separately verify integration with gem5's devices and state transfer. These experiments show that we transfer state correctly, but also uncovers several functional bugs in gem5's simulated CPUs. To evaluate the accuracy of our proposed sampling scheme, we compare the results of a traditional, non-sampling, reference simulation of the first 30 billion instructions of the benchmarks to sampling using a gem5-based SMARTS implementation and pFSA. We show that pFSA can estimate the IPC of the simulated applications with an average error of 2.0%. To investigate sources of the error, we investigate the impact of cache warming on accuracy. Finally, we evaluate scalability in a separate experiment where we show that our parallel sampling method scales almost linearly up to 28 cores.

For our experiments we simulated a 64-bit x86 system (Debian Wheezy with Linux 3.2.44) with split 2-way 64 kB L1 instruction and data caches and a unified 8-way 2MB or 8MB L2 cache with a stride prefetcher. The simulated CPU uses gem5's OoO CPU model. See Table I for a summary of the important simulation parameters. We compiled all benchmarks with GCC 4.6 in 64-bit mode with x87 code generation disabled[2]. We evaluated the system using the SPEC CPU2006 benchmark suite with the reference data set and the SPEC runtime harness. All simulation runs were started from the same checkpoint of a booted system. Simulation execution rates are shown running on a 2.3 GHz Intel Xeon E5520.

Pipeline	gem5's default OoO CPU	
	Store Queue	64 entries
	Load Queue	64 entries
Branch Predictors	Tournament Predictor	2-bit choice counters, 8 k entries
	Local Predictor	2-bit counters, 2 k entries
	Global Predictor	2-bit counters, 8 k entries
	Branch Target Buffer	4 k entries
Caches	L1I	64 kB, 2-way LRU
	L1D	64 kB, 2-way LRU
	L2	2 MB, 8-way LRU, stride prefetcher

Table I: Summary of simulation parameters.

SMARTS, FSA, and pFSA all use a common set of parameters controlling how much time is spent in their different execution modes. In all sampling techniques, we executed 30 000 instructions in the detailed warming mode and 20 000 instructions in the detailed sampling mode. The length of detailed warming was chosen according to the method in the original SMARTS work[2] and ensures that the OoO pipeline is warm. Functional warming for FSA and pFSA was determined heuristically to require 5 million and 25 million instructions for the 2 MB L2 cache and 8 MB L2 cache configurations, respectively. Methods to automatically select appropriate functional warming have been proposed[13], [14] by other authors and we outline a method leveraging our warming error estimates in the future work section. Using these parameters, we took 1000 samples per benchmark. Due to the slow reference simulations, we limit accuracy studies to the first 30 billion instructions from each of the benchmarks, which corresponds to roughly a week's worth of simulation time in the OoO reference. For these cases, the sample period was adjusted to ensure 1000 samples in the first 30 billion instructions.

A. *Validating Functional Correctness*

For a simulation study to be meaningful, we need to be confident that instructions executed in the simulator produce the right results, i.e., they are functionally correct. Incorrect execution can result in anything from subtle behavior changes to applications crashing in the simulated system. To assess correctness of our gem5 extensions, we rely on SPEC's built-in verification harness, which compares the output of a benchmark to a reference[3]. In order to use this harness, we execute all benchmarks to completion with their reference data sets. We verify that our gem5 additions to support hardware virtualization work correctly by running the benchmarks solely on the virtual CPU module (devices are still simulated by gem5). This experiment ensures that basic features, such as the interaction between the memory and simulated device models work correctly, and that the experimental setup (compilers, OS, and SPEC) is correct. We then verify that our reference

[2] We disabled x87 code generation in the compiler, forcing it to generate SSE code instead, since the simulated gem5 CPUs only support a limited number of x87 instructions.

[3] We realize that this is not sufficient to guarantee functional correctness, but we use SPEC's verification suite here since it is readily available.

simulations are correct by completing and verifying them using VFF.

In the first experiment, we verify that the virtual CPU module works correctly, including its interactions with the virtualized hardware and the simulated system in gem5. To do this, we execute and verify all benchmarks using only the virtual CPU module. In this experiment, all benchmarks executed to completion and completed their verification runs successfully. This demonstrates that: a) our virtual CPU module interacts correctly with the memory system and device models in gem5, and, b) our simulated system and the benchmark setup are working correctly.

In the second experiment, we evaluate the correctness of our reference simulations. In this experiment, we simulated all benchmarks for 30 billion instructions using the detailed CPU module and ran them to completion using VFF. This experiment showed that 9 out of the 29 benchmarks failed before finishing the simulation of the first 30 billion instructions and that another 7 failed to verify after running to completion. In order to verify that the benchmarks that failed after executing the first 30 billion instructions were not caused by incorrect state transfer between the simulated CPU and the virtual CPU module, we set up another experiment where we switched each benchmark 300 times between the simulated CPU and the virtual CPU module. In this experiment, all benchmarks, with the exception of 447.dealII (which failed because of unimplemented instructions), ran to completion and verified. The results of these experiments are summarized in Table II. Their results indicate that our virtual CPU module works and transfers state correctly. Unfortunately, they also indicate that the x86 model in gem5 still has some functional correctness issues (in our experience, both the Alpha and ARM models are much more reliable).

Since benchmarks that do not verify take different program paths in the simulator and on real hardware, we exclude them from the rest of the evaluation.

B. Accuracy

A simulator needs to be accurate in order to be useful. The amount of accuracy needed depends on which question the user is asking. In many cases, especially when sampling, accuracy can be traded off for performance. In this section, we evaluate the accuracy of our proposed parallel sampling methodology. The sampling parameters we use have been selected to strike a balance between accuracy and performance when estimating the average CPI of an application.

All sampling methodologies that employ functional warming suffer from two main sources of errors: sampling errors and inaccurate warming. Our SMARTS and pFSA experiments have been setup to sample at the same instructions counts, which implies that they should suffer from the same sampling error[4]. Functional warming incurs small variations in the access streams seen by branch predictors and caches since it does not include effects of speculation or reordering. This has can lead to a small error, which has been shown[2] to be in the region of 2%. The error incurred by these factors is the baseline SMARTS error, which in our experiments is 1.87% for a 2 MB L2 cache and 1.18% for an 8 MB L2 cache.

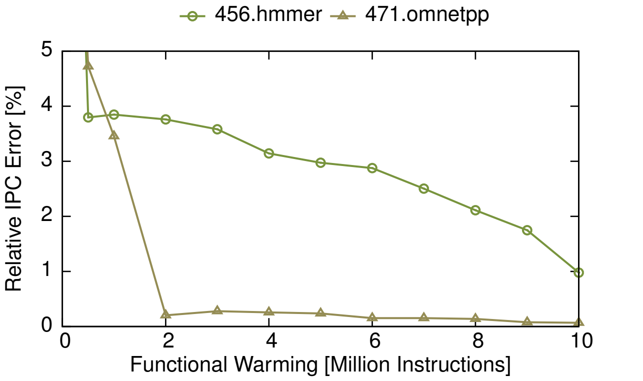


Figure 4: Estimated relative IPC error due to insufficient cache warming as a function of functional warming length for 456.hmmer and 471.omnetpp.

Another source of error is the limited functional warming of branch predictors and caches in FSA and pFSA. In general, our method provides very similar results to our gem5-based SMARTS implementation. However, there are a few cases (e.g., 456.hmmer) when simulating a 2 MB cache where we did not apply enough warming. In these case the IPC predicted by SMARTS is within, or close to, the warming error estimated by our method (Figure 3a). A large estimated warming error generally indicate that a benchmark should have had more functional warming applied. Note that we can not just assume the higher IPC since some warming misses are likely to be real misses.

To better understand how warming affects the predicted IPC bound, we simulated two benchmarks (456.hmmer & 471.omnetpp) with different warming behaviors with different amounts of cache warming. Figure 4 shows how their estimated warming error relative to the IPC of the reference simulations shrinks as more cache warming is performed. These two applications have wildly different warming behavior. While 471.omnetpp only requires two million instructions to reach an estimated warming error less than 1%, 456.hmmer requires more than 10 million instructions to reach the same goal.

C. Performance & Scalability

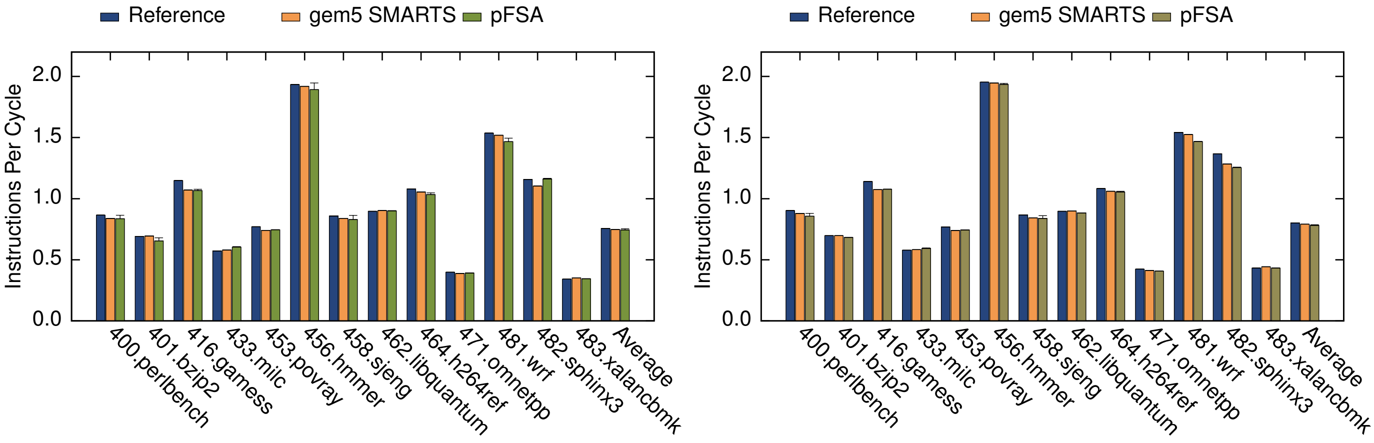
A simulator is generally more useful the faster it is as high speed enables greater application coverage and quicker simulation turn-around times. Figure 5 compares the execution rates of native execution, VFF, FSA, and pFSA when simulating a system with a 2MB and 8MB last-level cache. The reported performance of pFSA does not include warming error estimation, which adds 3.9% overhead on average. The achieved simulation rate of pFSA depends on three factors. First, fast-forwarding using VFF runs at near-native (90% on average) speed, which means that the simulation rate of an application is limited by its native execution rate regardless of parallelization. Second, each sample incurs a constant cost. The

[4] There might be slight differences when the virtual CPU module is used due to small timing differences when delivering asynchronous events (e.g., interrupts).

Benchmark			Verifies in Reference	Verifies using VFF	Verifies when Switching
400.perlbench	401.bzip2	416.gamess	Yes	Yes	Yes
433.milc	453.povray	456.hmmer			
458.sjeng	462.libquantum	464.h264ref			
471.omnetpp	481.wrf	482.sphinx3			
483.xalancbmk					
410.bwaves	434.zeusmp	435.gromacs	**No**	Yes	Yes
436.cactusADM	444.namd	459.GemsFDTD			
470.lbm					
445.gobmk	450.soplex	454.calculix	**Fatal Error**[1]	Yes	Yes
429.mcf	473.astar		**Fatal Error**[2]	Yes	Yes
437.leslie3d			**Fatal Error**[3]	Yes	Yes
403.gcc			**Fatal Error**[4]	Yes	Yes
447.dealII			**Fatal Error**[5]	Yes	**No**
465.tonto			**Fatal Error**[6]	Yes	Yes
Summary:			13/29 verified, 9/29 fatal	29/29 verified	28/29 verified

1. Simulator gets stuck.
2. Triggers a memory leak causing the simulator crash.
3. Terminates prematurely for unknown reason.
4. Fails with internal error. Likely due to unimplemented instructions.
5. Benchmark segfaults due to unimplemented instructions.
6. Terminated by internal benchmark sanity check.

Table II: Summary of verification results for all benchmarks in SPEC CPU2006. This table is based on three experiments: a reference OoO simulation that is completed using the virtual CPU module, purely running on the virtual CPU module, and repeatedly switching between a simulated OoO CPU and the virtual CPU module.



(a) 2 MB L2

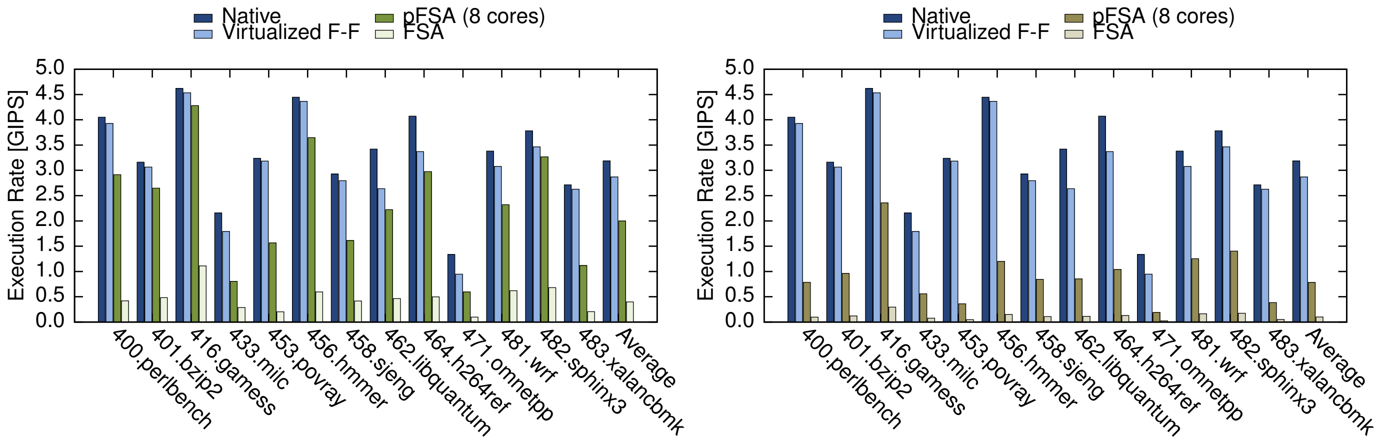
(b) 8 MB L2

Figure 3: IPC for the first 30 billion instructions of each benchmark as predicted by a reference simulation compared to a our gem5-based SMARTS implementation and pFSA. The error bars extending from the pFSA bars represent warming error estimates.

longer a benchmark is, the lower the average overhead. Third, large caches need more functional warming, and the longer the functional warming, the greater the cost of the sample. As seen when comparing the average simulation rates for a 2 MB cache and an 8 MB cache, simulating a system with larger caches incurs a larger overhead.

The difference in functional warming length results in different simulation rates for 2MB and 8MB caches. While the 8MB cache simulation is slower to simulate than the smaller cache, there is also more parallelism available. Looking at the simulation rate when simulating a 2MB cache as a function of the number of threads used by the simulator (Figure 6) for a fast (416.gamess) and a slow (471.omnetpp) application, we see that both applications scale almost linearly until they reach 93% and 45% of native speed respectively. The larger cache on the other hand starts off at a lower simulation rate and scales linearly until all cores in the host system are occupied. We estimate the overhead of copying simulation state (Fork Max) by removing the simulation work in the child and keeping the child process alive to force the parent process to do CoW while fast-forwarding. This is an estimate of the speed limit imposed by parallelization overheads.

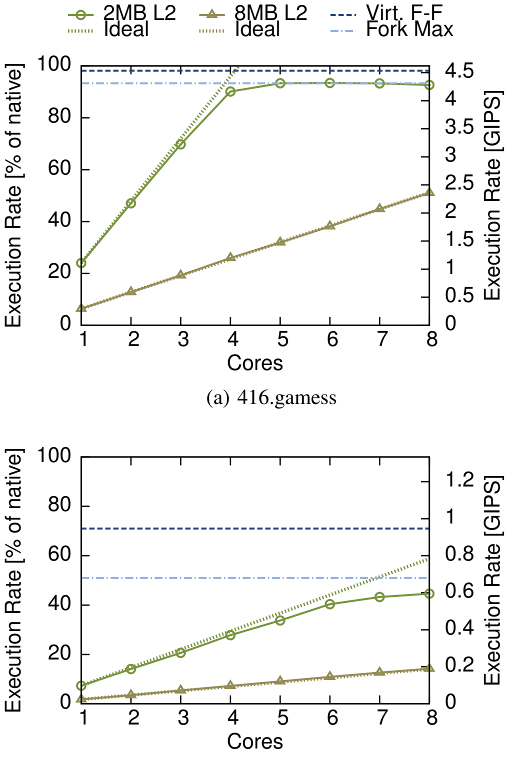
In order to understand how pFSA scales on larger systems, we ran the scaling experiment on a 4-socket Intel Xeon E5-



(a) 2 MB L2

(b) 8 MB L2

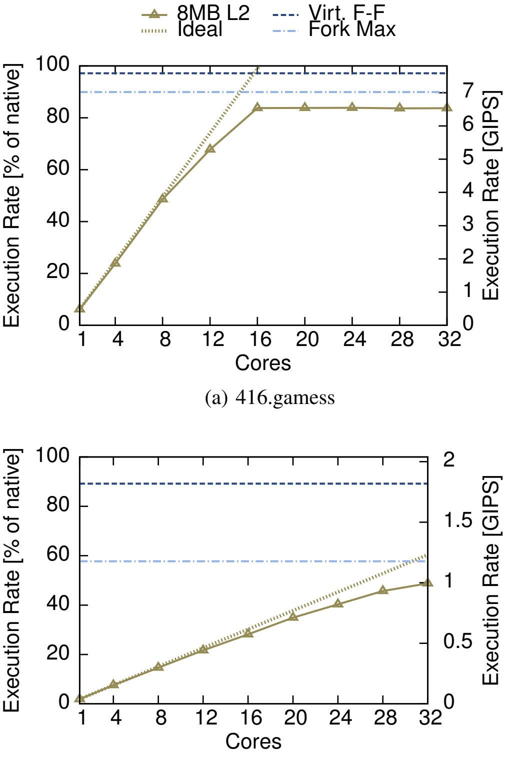
Figure 5: Execution rates when simulating a 2MB (a) and 8MB (b) L2 cache for pFSA and FSA compared to native and fast-forwarding using the virtual CPU module.



(a) 416.gamess

(b) 471.omnetpp

Figure 6: Scalability of 416.gamess (a) and 471.omnetpp (b) running on an 2-socket Intel Xeon E5520.



(a) 416.gamess

(b) 471.omnetpp

Figure 7: Scalability of 416.gamess (a) and 471.omnetpp (b) running on a 4-socket Intel Xeon E5-4650.

4650 with a total of 32 cores. We limited this study to the 8MB cache since simulating a 2MB cache reached near-native speed with only 8 cores. As seen in Figure 7, both 416.gamess and 471.omnetpp scale almost linearly until they reach their maximum simulation rate, peaking at 84% and 48.8% of native speed, respectively.

VI. Related Work

Our parallel sampling methodology builds on ideas from three different simulation and modeling techniques: virtualization, sampling, and parallel profiling. We extend and combine ideas from these areas to form a fully-functional, efficient, and scalable full-system simulator using the well-known gem5[1] simulation framework.

A. Virtualization

There have been several earlier attempts at using virtualization for full-system simulation. Rosenblum et al. pioneered the use of virtualization-like techniques with SimOS[15] that ran a slightly modified (relinked to a non-default memory area) version of Irix as a UNIX process and simulated privileged instructions in software. PTLsim[5] by Yourst, used para-virtualization[5] to run the target system natively. Due to the use of para-virtualization, PTLsim requires the simulated operating system to be aware of the simulator. The simulated system must therefore use a special para-virtualization interface to access page tables and certain low-level hardware. This also means that PTLsim does not simulate low-level components like timers and storage components (disks, disk controllers, etc.). A practical draw-back of PTLsim is that it practically requires a dedicated machine since the host operating system must run inside the para-virtualization environment. Both SimOS and PTLsim use a fast virtualized mode for fast-forwarding and support detailed processor performance models. The main difference between them and gem5 with our virtual CPU module is the support for running unmodified guest operating systems. Additionally, since we only depend on KVM, our system can be deployed in shared clusters with unmodified host operating systems.

An interesting new approach to virtualization was taken by Ryckbosch et al. in their VSim[16] proposal. This simulator mainly focuses on IO modeling in cloud-like environment. Their approach employs time dilation to simulate slower or faster CPUs by making interrupts happen more or less frequently relative to the instruction stream. Since the system lacks a detailed CPU model, there are no facilities for detailed simulation or auto-calibration of the time dilation factor. In many ways, the goals of VSim and pFSA are very different: VSim focuses on fast modeling of large IO-bound workloads, while pFSA focuses on sampling of detailed micro-architecture simulation.

B. Sampling

Techniques for sampling simulation have been proposed many times before[2], [9], [10], [11], [3], [17], [18]. The two main techniques are SimPoint[3] and SMARTS[2]. While both are based on sampling, SimPoint uses a very different approach compared SMARTS and pFSA that builds on checkpoints of representative regions of an application. Such regions are automatically detected by finding phases of stable behavior. In order to speed up SMARTS, Wenisch et al. proposed TurboSMARTS[9], which uses compressed checkpoints that include cache state and branch predictor state. A drawback of all checkpoint-based techniques is long turn-around time if the *simulated software* changes due to the need to collect new checkpoints. This makes them particularly unsuitable for many applications, such as hardware-software co-design or operating system development. Since pFSA uses virtualization instead of checkpoints to fast-forward between samples, there is no need to perform costly simulations to regenerate checkpoints when making changes in the simulated system.

SMARTS has the nice property of providing statistical guarantees on sampling accuracy. These guarantees assure users who strictly follow the SMARTS methodology that their sampled IPC will not deviate more than, for example, 2% with 99.7% confidence. Since we do not perform always-on cache and branch predictor warming, we can not provide the same statistical guarantees, but we achieve similar accuracy in practice. To identify problems with insufficient warming, we have proposed a low-overhead approach that can estimate the warming error.

The sampling approach most similar to FSA is the one used in COTSon [18] by HP Labs. COTSon combines AMD SimNow[19] (a JIT:ing functional x86 simulator) with a set of performance models for disks, networks, and CPUs. The simulator achieves good performance by using a dynamic sampling strategy [17] that uses online phase detection to exploit phases of execution in the target. Since the functional simulator they use can not warm microarchitectural state, they employ a two-phase warming strategy similar to FSA. However, unlike FSA, they do not use hardware virtualization to fast-forward execution, instead they rely on much slower (10x overhead [18] compared to 10% using virtualization) functional simulation.

C. Parallel Simulation

There have been many approaches to parallelizing simulators. We use a coarse-grained high-level approach in which we exploit parallelism between samples. A similar approach was taken in SuperPin[6] and Shadow Profiling[7], which both use Pin[8] to profile user-space applications and run multiple parts of the application in parallel. Shadow Profiling aims to generate detailed application profiles for profile guided compiler optimizations, while SuperPin is a general-purpose API for parallel profiling in the Pin instrumentation engine. Our approach to parallelization draws inspiration from these two works and uses parallelism to overlap detailed simulation of multiple samples with native execution. The biggest difference is that we apply the technique to full-system simulation instead of user-space profiling.

Another approach to parallelization is to parallelize the core of the simulator. A significant amount of research has been done on parallel discrete event simulation (PDES), each proposal with its own trade-offs[20]. *Optimistic approaches* try to run as much as possible in parallel and roll-back whenever there is a conflict. Implementing such approaches can be challenging since they require old state to be saved. *Conservative approaches* typically ensure that there can never be conflicts by synchronizing at regular intervals whose length is determined by the shortest critical path between two components simulated in parallel. The latter approach was used in the Wisconsin Wind Tunnel[21]. More recent systems, for example Graphite[22], relax synchronization even further. They exploit the observation that functional correctness is not affected as long as synchronization instructions (e.g., locks) in the simulated system enforce synchronization between simulated

[5] There have been signs of an unreleased prototype of PTLSim that supports hardware virtualization. However, to the best of our knowledge, no public release has been made nor report published.

threads. The amount of drift between threads executed in parallel can then be configured to achieve a good trade-off between accuracy and performance.

The recent ZSim[23] simulator takes another fine-grained approach to parallelize the core of the simulator. ZSim simulates applications in two phases, a *bound* and a *weave* phase, the phases are interleaved and only work on a small number of instructions at a time. The bound phase executes first and provides a lower bound on the latency for the simulated block of instructions. Simulated threads can be executed in parallel since no interactions are simulated in this phase. The simulator then executes the weave phase that uses the traces from the bound phase to simulate memory system interactions. This can also be done in parallel since the memory system is divided into domains with a small amount of communication that requires synchronization. Since ZSim is Pin-based, ZSim only supports user-space x86 code and does not simulate any devices (e.g., storage and network). The main focus of ZSim is simulating large parallel systems.

Methods such as PDES or ZSim are all orthogonal to our pFSA method since they work at a completely different level in the simulator. For examples, a simulator using PDES techniques to simulate in parallel could be combined with pFSA to expose even more parallelism than can be exposed by PDES alone.

VII. Future Work

There are several features and ideas we would like to explore in the future. Most notably, we would like add support for running multiple virtual CPUs at the same time in a shared-memory configuration when fast-forwarding. KVM already supports executing multiple CPUs sharing memory by running different CPUs in different threads. Implementing this in gem5 requires support for threading in the core simulator, which is ongoing work from other research groups. We are also looking into ways of extending warming error estimation to TLBs and branch predictors. An interesting application of warming estimation is to quickly profile applications to automatically detect per-application warming settings that meet a given warming error constraint. Additionally, an online implementation of dynamic cache warming could use feedback from previous samples to adjust the functional warming length on the fly and use our efficient state copying mechanism to roll back samples with too short functional warming.

VIII. Summary

In this paper, we have presented a virtualized CPU module for gem5 that on average runs at 90% of the host's execution rate. This CPU module can be used to efficiently fast-forward simulations to efficiently create checkpoints of points of interest or to implement efficient performance sampling. We have demonstrated how it can be used to implement an efficient parallel sampler, pFSA, which accurately (IPC error of 2.2% and 1.9% when simulating 2MB and 8MB L2 caches respectively) estimates application behavior with high performance (63% or 25% of native depending on cache size). Compared to detailed simulation, our parallel sampling simulator results in 7 000x–19 000x speedup.

Acknowledgments

Initial work on hardware virtualization support in gem5 was sponsored by ARM, where the authors would especially like to thank Matthew Horsnell, Ali G. Saidi, Andreas Hansson, Marc Zyngier, and Will Deacon for valuable discussions and insights. Reference simulations were performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX). This work was financed by the CoDeR-MP project and the UPMARC research center.

References

[1] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. A. Wood, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, and T. Krishna, "The gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, Aug. 2011.

[2] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," in *Proc. International Symposium on Computer Architecture (ISCA)*, 2003, pp. 84–95.

[3] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," in *Proc. Internationla Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002, pp. 45–57.

[4] A. Kivity, U. Lublin, and A. Liguori, "kvm: the Linux Virtual Machine Monitor," in *Proc. Linux Symposium*, 2007, pp. 225–230.

[5] M. T. Yourst, "PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator," in *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*, Apr. 2007, pp. 23–34.

[6] S. Wallace and K. Hazelwood, "SuperPin: Parallelizing Dynamic Instrumentation for Real-Time Performance," in *Proc. International Symposium on Code Generation and Optimization (CGO)*, Mar. 2007, pp. 209–220.

[7] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri, "Shadow Profiling: Hiding Instrumentation Costs with Parallelism," in *Proc. International Symposium on Code Generation and Optimization (CGO)*. IEEE, Mar. 2007, pp. 198–208.

[8] 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 *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[9] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe, "TurboSMARTS: Accurate Microarchiteecture Simulation Sampling in Minutes," *ACM SIGMETRICS Performance Evaluation Review*, vol. 33, no. 1, pp. 408–409, Jun. 2005.

[10] S. Chen, "Direct SMARTS: Accelerating Microarchitectural Simulation through Direct Execution," Master's thesis, Carnegie Mellon University, 2004.

[11] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "SimFlex: Statistical Sampling of Computer System Simulation," *IEEE Micro*, vol. 26, no. 4, pp. 18–31, Jul. 2006.

[12] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, pp. 92–99, 2005.

[13] Y. Luo, L. K. John, and L. Eeckhout, "Self-Monitored Adaptive Cache Warm-Up for Microprocessor Simulation," in *Proc. Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2004, pp. 10–17.

[14] M. Van Biesbrouck, B. Calder, and L. Eeckhout, "Efficient Sampling Startup for SimPoint," *IEEE Micro*, vol. 26, no. 4, pp. 32–42, Jul. 2006.

[15] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta, "Complete Computer System Simulation: The SimOS Approach," *Parallel & Distributed Technology: Systems & Applications*, vol. 3, no. 4, pp. 34–43, Jan. 1995.

[16] F. Ryckbosch, S. Polfliet, and L. Eeckhout, "VSim: Simulating Multi-Server Setups at Near Native Hardware Speed," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, pp. 52:1–52:20, 2012.

[17] A. Falcón, P. Faraboschi, and D. Ortega, "Combining Simulation and Virtualization through Dynamic Sampling," in *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*, Apr. 2007, pp. 72–83.

[18] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "COTSon: Infrastructure for Full System Simulation," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 1, pp. 52–61, Jan. 2009.

[19] R. Bedicheck, "SimNow$^{\text{TM}}$: Fast Platform Simulation Purely in Software," in *Hot Chips: A Symposium on High Performance Chips*, Aug. 2005.

[20] R. M. Fujimoto, "Parallel Discrete Event Simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, Oct. 1990.

[21] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood, "The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers," *ACM SIGMETRICS Performance Evaluation Review*, vol. 21, no. 1, pp. 48–60, Jun. 1993.

[22] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A Distributed Parallel Simulator for Multicores," in *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*, Jan. 2010, pp. 1–12.

[23] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *Proc. International Symposium on Computer Architecture (ISCA)*, Jul. 2013, pp. 475–486.