SimuBoost: Scalable Parallelization of Functional System Simulation

Marc Rittinghaus Konrad Miller Marius Hillenbrand Frank Bellosa

System Architecture Group
Karlsruhe Institute of
Technology (KIT)
firstName.lastName@kit.edu

ABSTRACT

The limited execution speed of current full system simulators restricts their applicability for dynamic analysis to short-running workloads. When analyzing memory contents while simulating a kernel build with Simics, we encountered slowdowns of more than 5000x resulting in 10 months of total simulation time. Prior work improved the simulation speed by simulating virtual CPU cores on separate physical CPU cores simultaneously or by applying sampling and extrapolation methods to focus costly analyses on short execution windows. However, these approaches inherently suffer from limited scalability or trading accuracy for speed. SimuBoost is a novel idea to parallelize functional full system simulation of single-cores. Our approach takes advantage of fast execution through virtualization, taking checkpoints in regular intervals. The parts between subsequent checkpoints are then simulated and analyzed simultaneously in one job per interval. By transferring jobs to multiple nodes, a parallelized and distributed simulation of the target workload can be achieved, thereby effectively reducing the overall required simulation time. As no implementation of SimuBoost exists yet, we present a formal model to evaluate the general speedup and scalability characteristics of our acceleration technique. We moreover provide a model to estimate the required number of simulation nodes for optimal performance. According to this model, our approach can speed up conventional simulation in a realistic scenario by a factor of 84, while delivering a parallelization efficiency of 94%.

Categories and Subject Descriptors

D.4.8 [**Operating Systems**]: Performance—*Simulation*

Keywords

Full System Simulation, Dynamic Performance Analysis, Virtualization, Checkpointing, Virtual Machine Replay, Parallelization, Operating System Analysis

1. INTRODUCTION

Full system simulation allows simulating an entire physical machine on top of a host operating system (OS) and thus provides a powerful foundation to study the runtime behavior and interaction of computer architecture, operating systems and applications [1, 21, 29]. Since the entire execution environment in such a system is virtual, every operation carried out can be inspected easily. In contrast to the instrumentation of applications or the OS, analysis with a simulator does not influence the simulated machine's state and thus can be arbitrarily complex in time and space without distorting measurements. While the focus of *functional simulation* (a.k.a. *emulation*) lies in preserving functional correctness, *timing models* enhance the simulation accuracy by incorporating operation latency of simulated devices. *Micro-architectural simulation* further increases precision through detailed processor- and device state models.

A well-known limitation of full system simulation is the low execution speed offered by current simulators. Table 1 gives an overview of the single-core execution speed in million instructions per second (MIPS), the time to completion (TTC) in hours as well as the slowdown ($s_{virt/sim}$) for various benchmarks. Each workload has been run natively, in a virtual machine (VM) using KVM [10], and simulated with QEMU [4] and Simics [14], respectively. Observing operations in a simulator slows down the execution but is essential to perform analyses. We therefore installed memory access hooks in both simulators to measure a realistic simulation speed for a typical memory analysis scenario.

The slowdown incurred by functional simulation compared to hardware-assisted virtualization is significant (31x-810x on average). That quickly renders functional simulation impractical for long-running workloads (e.g., 50 days for SPEC CPU2006). The extra slowdown for the second set of hooks moreover shows that the execution time is very sensitive to additional overhead. *Representative sampling* [23] can reduce the run-time overhead by limiting complex analyses to short time frames that are representative for the analyzed workload. However, an initial functional simulation to identify such intervals is still needed and the accuracy achievable with this technique also heavily depends on sufficient phase behavior in the workload, which is not always present [27]. Moreover, in some scenarios (e.g., analysis of memory duplication) limiting the observation window is not an option. An acceleration technique to enable full-length analyses of long-running workloads is thus desirable.

	Bare Metal	Hw-Virt. KVM	Simulation QEMU[1]	QEMU[2]	Simics[1]
Linux 3.7.1 Kernel Build					
MIPS	2314	2122	71	14	3
TTC [h]	1.4	1.6	46.9	238	1080
$s_{virt/sim}$	–	≈ 1	≈ 33	≈ 165	≈ 771
SPECint_base2006 1.2					
MIPS	3108	2989	140	15	3
TTC [h]	6	6.3	133.2	1243.2	6216
$s_{virt/sim}$	–	≈ 1	≈ 22	≈ 207	≈ 1036
LAMMPS Lennard Jones					
MIPS	2495	2642	65	22	4
TTC [h]	1.8	1.7	69.1	204.1	1123
$s_{virt/sim}$	–	≈ 1	≈ 38	≈ 113	≈ 624

[1] Empty hooks without analysis overhead.
[2] Hooks measuring accessed physical pages per second.

Table 1: Measurements were taken on a dual socket Xeon E-2420 system, virtualizing/simulating a single-core VM. The native execution was restricted to a single core, accordingly. Functional simulation incurs an avg. slowdown of $s_{sim} \approx 31$ for QEMU and $s_{sim} = 810$ for Simics. Performing analyses heavily reduces execution speed further (for QEMU in our example $s_{sim} \approx 162$).

SimuBoost strives to close the performance gap between virtualization and functional simulation. The core idea is to run the workload in a VM, taking checkpoints in regular intervals. The parts between subsequent checkpoints are then simulated and analyzed simultaneously in one job per interval. By transferring jobs to multiple nodes, a parallelized and distributed simulation of the target workload can be achieved, thereby reducing the overall simulation time.

As we are still working on the implementation of SimuBoost, we cannot provide empirical results, yet. Instead we give a first evaluation of the practical feasibility of our approach by presenting a formal model to describe its speedup and scalability characteristics. SimuBoost can speed up conventional simulation in a realistic scenario (parameter-wise) by a factor of 84, while delivering a parallelization efficiency of 94% according to the model.

The remainder of this paper is structered as follows: Our approach to parallelize functional simulation is described in Section 2. Section 3 discusses the practical feasibility and limitations of our acceleration technique. An overview of prior work on acceleration of full system simulation is provided in Section 4. We conclude and give a prospect on future work in Section 5.

2. APPROACH

Our approach delivers a method for functional simulation that provides significantly faster execution than current implementations and that makes inspecting the full run-time feasible even for long-running workloads. Current acceleration techniques for functional full system simulations (a) limit costly inspections to short time frames, trading accuracy for speed or (b) do not scale beyond the simulated parallelism. SimuBoost addresses these drawbacks.

The core idea of our acceleration technique is to split the simulation time into independent intervals that can be simulated simultaneously with conventional functional simulation (see Figure 1). The benefit of taking this approach as foundation lies in the fact that it scales with the *run-time* of the simulation: the longer the simulation (including analysis) takes, the more intervals can be extracted and the higher is the degree of parallelization. Opposed to approaches that map simulated CPU cores in a multi-core simulation to real parallelism in the host [6, 12, 26, 31], splitting the simulation into intervals does not limit the degree of parallelization to the number of simulated cores. This way our method is applicable even to single-core simulations. Moreover, since the intervals can be processed independently, it allows distributing the simulation workload across multiple hosts. To match the number of intervals to the available hardware resources the interval length needs to be chosen accordingly.

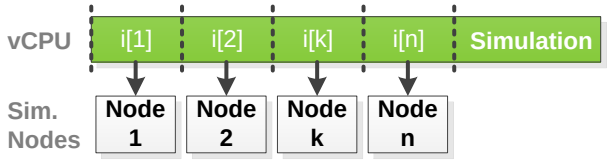

Figure 1: The simulation is split along the time axis into n intervals; with n being the required degree of parallelism. Each interval is simulated on a different node (i.e., CPU core, host, etc.).

A fundamental challenge with this approach is that every interval $i[k]$ depends on the execution of the previous interval $i[k-1]$. The simulation of $i[k]$ thus cannot be started in advance without knowing the simulated machine's state at the beginning of $i[k]$. We solve this problem by utilizing virtualization as a fast-forward execution mode to collect state information as quickly as possible on-the-fly (see Figure 2).

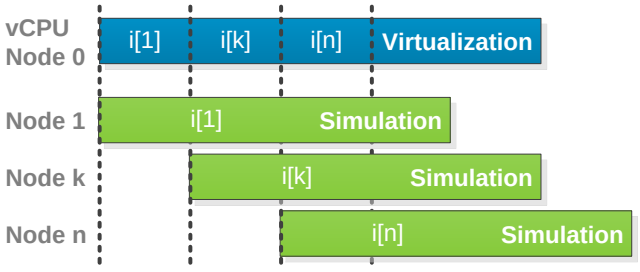

Figure 2: The workload is executed with virtualization. Checkpoints at the interval boundaries serve as starting points for parallel simulations.

The virtual machine that is to be inspected runs on a dedicated node managed by a hypervisor such as KVM [10]. At each interval boundary the hypervisor takes a snapshot of the full system state (i.e., memory content, device states, HDD data, etc.). The checkpoints then serve as starting points for simulations. Consequently, although each simulation is delayed up to the point when the respective interval is reached in the virtualization stage, the execution speed difference between virtualization and functional simulation enables a parallelization of the simulation.

State Deviation

Previous work uses deterministic functional simulation to fast-forward to the point in execution that is to be analyzed [2, 12, 14, 16, 18]. In such an environment, splitting the simulation into intervals is feasible, because at the end of an interval the deterministic execution delivers a machine state that exactly matches the one representing the starting point for the subsequent interval. This is also true if intervals are simulated with different degrees of detail (e.g., functional vs. micro-architectural simulation). DiST [8] bases its acceleration technique on this property. However, it does not apply to our proposed combination of virtualization and simulation. Taking advantage of non-deterministic virtualization imposes a new challenge regarding the synchronization of fast- and slow execution modes: The execution through virtualization is directly influenced by non-deterministic hardware behavior such as varying disk response times. Figure 3 illustrates this for non-deterministic I/O completion:

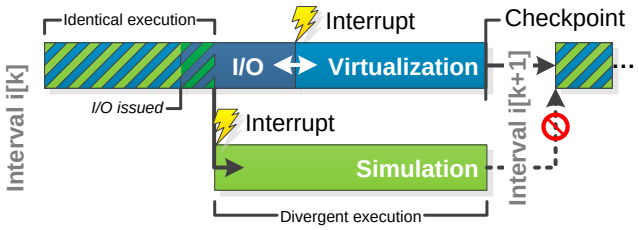

Figure 3: Non-deterministic behavior in the virtualization causes the VM states to drift apart, leading to a break in continuity for the simulation stage at interval boundaries.

Since the I/O completion time with virtualization is unknown in advance, interrupt delivery in the virtualization and the simulation of the respective interval take place at different times. In consequence, the subsequent execution in both environments drifts apart. This poses a problem at the end of an interval, because the machine state in the simulation does not match the one in the virtualization. Starting the next interval's simulation based on the checkpoint created by the virtualization, therefore, leads to a break in state continuity from the simulation's perspective. As the machine state produced by the simulation is the one that is analyzed by the user, a disruption in continuity is highly probable to corrupt measurements. Furthermore, analyzing a simulation that suffers from state deviation is of questionable use as its execution does not match the observed behavior in the VM.

Non-deterministic delays in the hardware are not the only source of state deviation. External input (e.g., network traffic, user input, etc.), instructions such as Read Time Stamp Counter (rdtsc) and the output of special devices (e.g., random number generators) also lead to state deviation.

Coping with State Deviation

State deviation is caused by non-deterministic events in the virtualization that do not equally appear in the simulation. To avoid deviation, those events need to be brought into line across both execution modes. This can be accomplished by precisely replaying each event in the simulation (see Figure 4).

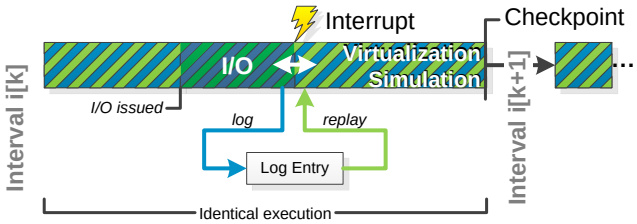

Figure 4: Non-deterministic events are logging in the hypervisor for an replay within the simulator.

The hypervisor traps each occurrence of non-deterministic actions in the virtualization and logs enough information to allow the simulator replaying the event. Interrupt delivery for instance needs to be delayed in the simulation up to the exact execution point (instruction granularity) where it appeared in the virtualization. For non-deterministic events that include unpredictable input such as keyboard input or the result of special instructions, the hypervisor additionaly needs to capture the corresponding data.

The replay ensures that the instruction stream in both execution environments matches and prevents a break of continuity in the simulation stage. However, to be able to access logging information, an interval's simulation must be delayed until its execution finishes in the virtualization and all events have been fully recorded. Hence, the simulation of interval $i[k]$ can be started when the hypervisor creates the checkpoint for interval $i[k+1]$.

3. DISCUSSION

To support our approach, we first demonstrate its technical feasibility with regard to its performance requirements and run-time overhead. To this end, we present numbers reported in related work. We then elaborate on the speedup and scalability characteristics of our approach by providing a formalization that allows estimating the benefit of SimuBoost. We conclude the discussion with an overview of limitations present in the current design.

Feasibility

The feasibility of our approach strongly depends on the ability to (a) log and replay non-deterministic events (this way, avoiding state deviation) and (b) to keep a high virtualization speed despite active event logging and checkpointing.

Dunlap et al. already developed a comparable logging mechanism with ReVirt [7] to enable intrusion analysis through virtual machine logging and replay. Their evaluation revealed that an exact replay of a virtual machine is feasible. Their logging mechanism shows a run-time overhead of 0-8% and a storage space requirement of up to 60 MiB per hour of workload run-time. Retrace [22] is a trace collection tool that employs the same technique to do deterministic replay in the VMware hypervisor [25]. With 5% run-time overhead, it performs similarly to ReVirt.

Remus [24] offers an efficient checkpointing mechansim for the Xen [3] hypervisor. To achieve high availability for virtual machines Remus replicates VM checkpoints at very high frequencies. For a kernel build and 10 checkpoints per second, the authors of Remus measured a 32% performance penalty with a strong linear correlation between checkpoint-

ing rate and run-time overhead. As the frequency for SimuBoost can be lower (in the range of seconds), we estimate the overhead to be considerably smaller (e.g., below 5%).

To capture a consistent system state, a virtual machine must be suspended shortly for each checkpoint. The downtime introduced by this operation directly prolongs the run-time of the workload in the (serial) virtualization stage and consequently delays the launch of new simulations. It is therefore important to keep the downtime as short as possible. The authors of Remus report a VM downtime per checkpoint as low as 100 ms.

Speedup and Scalability

To illustrate the characteristics of our approach, we present a formal model that describes the most important metrics: These are T_{ps} := the total run-time of a parallelized simulation, L_{opt} := the optimal interval length for a given scenario, S_{opt} := the speedup over conventional serial simulation, and N_{opt} := the number of nodes required to achieve optimal speedup.

Figure 5 illustrates the parameters. For simplicity reasons, we assume the run-time of a single interval's simulation to be linearly proportional to L := the length of a single interval (i.e., an interval's run-time in the virtualization stage) with respect to s_{sim} := the effective slowdown between virtualization and functional simulation—including analysis overhead. We moreover assume the last interval to be simulated on the virtualization node.

Let n := the number of intervals, t_c := the constant VM downtime for a checkpoint and t_i := a simulation's initialization time[1]. Further, let s_{log} := the slowdown in the virtualization stage incurred by the logging of non-deterministic events, T_{vm} := the workload's run-time with conventional virtualization and T_{sim} := the workload's run-time with conventional functional simulation. As a guiding example we choose $T_{vm} = 1\,\text{h} = 3600\,\text{s}$, $s_{sim} = 100$, $s_{log} = 1.08$, $t_c = 100\,\text{ms} = 0.1\,\text{s}$ and $t_i = 1\,\text{s}$. We can then express the total run-time of a parallelized simulation T_{ps} as follows:

$$T_{ps}(n) = s_{log}T_{vm} + n \cdot t_c + t_i + \frac{1}{n}T_{sim} \tag{1}$$

With $T_{sim} = s_{sim}T_{vm}$ and $n = \frac{s_{log}T_{vm}}{L}$, we can transform Equation 1 to express T_{ps} in terms of the interval length L, which is more useful in practice:

$$T_{ps}(L) = s_{log}T_{vm}(\frac{t_c}{L} + 1) + t_i + \frac{s_{sim}}{s_{log}}L \tag{2}$$

The speedup $S(L)$ of our approach compared to serial functional simulation is then:

$$\begin{aligned} S(L) &= \frac{T_{sim}}{T_{ps}(L)} \\ &= \frac{s_{log}T_{vm} \cdot s_{sim}L}{s_{log}^2 T_{vm}(t_c + L) + s_{log}t_i L + s_{sim}L^2} \end{aligned} \tag{3}$$

The speedup heavily depends on the chosen interval length L. Figure 6 illustrates this relationship. Although the potential degree of parallelization is higher for short intervals (a), the constant time overhead to take checkpoints at each interval boundary quickly becomes a limiting factor. With long intervals (c) the overhead caused by checkpoints is much lower. At the same time, a high interval length results in a higher per-interval run-time in the simulation stage. A high length thus also prolongs the simulation of the final interval and by that severely delays completion. The optimal interval length L_{opt} (b), on the other side, maximizes the speedup. To find it we solve $\frac{\partial}{\partial L}S(L) = 0$ for L. Since we can safely assume that all parameters are positive, we omit the negative result and get:

$$L_{opt} = \sqrt{\frac{s_{log}^2 T_{vm} t_c}{s_{sim}}} \tag{4}$$

$$S_{opt} = S(L_{opt}) \tag{5}$$

For our example, we thus have a speedup $S_{opt} \approx 84$ for an optimal interval length $L_{opt} \approx 2\,\text{s}$. To effectively achieve this speedup a sufficient number of simulation nodes must be supplied. Under the assumptions that the simulation of each interval takes equally long and that nodes take over the simulation of following intervals as soon as possible, we can calculate the required number of nodes N for a given interval length L as follows:

$$N(L) = \left\lceil \frac{t_i + \frac{s_{sim}}{s_{log}}L}{t_c + L} + 1 \right\rceil \tag{6}$$

$$N_{opt} = N(L_{opt}) \tag{7}$$

Figure 7 illustrates the rationale behind Equation 6. As new intervals are submitted, additional nodes are allocated until the first simulation finishes. After that, simulations can be

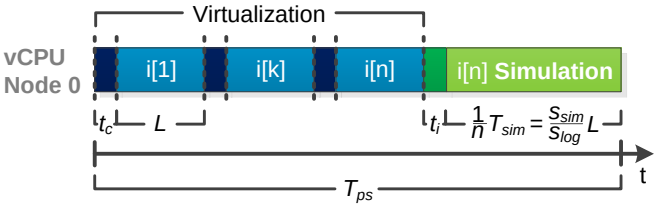

Figure 5: Overview of Parameters. The simulation of the last interval is scheduled on the virtualization node.

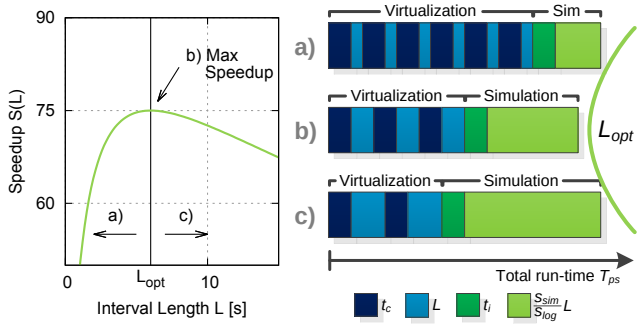

Figure 6: The right interval length is crucial for an optimal speedup (b). With too short intervals (a) the VM downtime dominates and the speedup rapidly decreases. Too long intervals (c) do not parallelize optimally.

[1]This is the time to (potentially) migrate the checkpoint to a different node, load it and initialize the simulated machine's state on the basis of the checkpointed information.

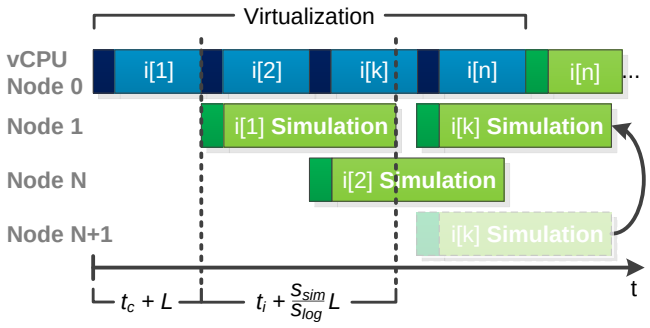

Figure 7: New simulation nodes are needed until the first interval simulation finishes. Subsequent intervals can be scheduled onto previously allocated nodes.

scheduled onto previously allocated nodes. This is because we assume simulations to complete with approximately the same rate than new intervals are submitted. In our example scenario, we have to supply $N_{opt} = 90$ cores for a speedup of 84. The efficiency of the parallelization can be expressed as follows:

$$E = \frac{S_{opt}}{N_{opt}} \tag{8}$$

In the example scenario we get an approximate efficiency of 94%. Hence, our approach is able to reduce the slowdown between virtualization and functional simulation from factor 100 down to around 1.19. In practice this means, that a workload that takes over three months to execute with conventional simulation, could complete in about a single day with SimuBoost.

Limitations

The dependency on virtualization as well as the changed execution environment introduce a few basic conditions that need to be considered:

Architecture. SimuBoost needs the virtualization stage to be as fast as possible, thus hardware support for virtualization of the target architecture should be present. As we are not aware of any hardware virtualization that is capable to accelerate an instruction set architecture (ISA) different from the one running the host OS, this restricts simulations to the host ISA. Due to the increased surface for state deviation in a parallel system and the difficulty to trap multi-core related events in the hypervisor (e.g., shared memory writes), we also have not considered simulating multi-core machines, yet.

Determinism. Like most of the acceleration approaches that parallelize multi-core simulations via per-vCPU dedicated hardware threads, our approach sacrifices strict determinism for speed. However, in contrast to previous approaches, determinism in the simulation stage remains preserved, because the simulation itself is unchanged. Repeated simulations based on the same set of checkpoints are thus executed equally. This leaves room to explore different parameters or methods for analyses on the exact same simulation. At the same time, the virtualization stage can provide a more realistic execution flow through the influence of non-deterministic hardware behavior.

Split Simulation. If the simulation modifies or creates state that is not transported through the VM checkpoints, intervals need to be overlapped to synchronize the states between subsequent intervals as proposed in DiST [8]. In the case of a cache simulation this means that both intervals would run simultaneously until their cache states are coherent. However, such cases are not covered in the current design. Moreover, splitting the simulation into intervals requires an analysis logic that supports independently processing ranges of the workload. Depending on the type of analysis performed this can complicate the logic. In general, analyses that fit into *MapReduce* semantic can be naturally run with our approach.

4. RELATED WORK

Table 2 lists popular full system simulators and some of the hardware platforms that are supported as simulation targets:

	Hardware Platforms	Sim. Level
Bochs [13]	x86-64	Functional
QEMU [4]	x86-64, ARM, MIPS...	Functional
Simics [14]	x86-64, ARM, MIPS...	Func. w. Timing
Embra [28]	MIPS	Func. w. Timing
COTson [2]	x86-64	Func. w. Timing
Mambo [5]	PowerPC	Func. w. Timing
SimOS [20]	MIPS, Alpha	Micro-Arch.
PTLSim [30]	x86-64	Micro-Arch.
MARSSx86 [18]	x86-64	Micro-Arch.

Table 2: Overview of Full System Simulators

Three major approaches to accelerate simulation have been proposed before:

Speed/Accuracy Trade-off. A commonly used practice to make analysis via full system simulation applicable to long-running workloads is to trade measuring accuracy for speed. To this end, many simulators support dynamically switching between different detailed simulation modes at run-time, thereby enabling the user to focus the analysis on short (representative) time frames. In Simics [14] the user is able to choose whether timing models are applied or not. MARSSx86 [18] similarly allows switching between functional and micro-architectural simulation. PTLSim/X [30] in contrast cooperates with Xen [3] to provide virtualization as an alternative mode of execution. Concentrating analysis efforts to certain windows, however, requires the use of sampling and extrapolation methods to generalize measurements. This increases the overall complexity and is difficult to accomplish if representative program regions are unknown. Moreover, depending on the use-case, it may be no option at all (e.g., debugging of race conditions).

Parallel Multi-Core Simulation. The simulation of multi-core machines becomes increasingly important as such systems are prevalent today. Many simulators officially or unofficially (i.e., via 3rd party patches) support parallelized execution by running each virtual CPU core on a dedicated hardware thread [6, 12, 26, 31]. While achieving good speedups (3.8x for a quad-core ARM simulation [6]), the approach does not accelerate the simulation of the virtual cores them-

selves. Its scalability is therefore inherently limited by the degree of simulated parallelism. At the same time, most implementations fully give up determinism by resorting to hardware arbitration in the host to order colliding memory writes. Graphite [16] is a many-core (i.e., thousands of cores) simulator, which takes parallelization a step further, distributing the simulation workload across multiple machines. Portero et al. expanded on these capabilities with a simulator that also delivers timing and functional models for on-chip interconnection systems [19].

Division of Simulation Time. The division of simulation time employed by our approach has already been proposed for accelerating micro-architectural simulation. Nguyen at al. proposed to use trace-driven simulation and to split the trace into separate time intervals that–in a second step–can be simulated in more detail simultaneously [17]. To generate the underlying instruction trace a preceding recording phase with functional simulation is utilized. Equally targeted at micro-architectural simulation, DiST [8] enhances the approach by providing a robust method to cope with the necessary model warm-up phase at the interval beginnings.

Further research has been invested into adjusting workloads to efficiently use the limited computational resources in full system simulations and to make finding promising windows in the evaluation parameter space easier [11]. There have also been attempts to optimize binary translated code [9, 15].

5. CONCLUSION

Leveraging virtualization to parallelize functional full system simulation provides new possibilities to study long-running workloads and to perform analyses that take too long with conventional serial simulation. We have presented an approach that is capable of accelerating functional simulation by two orders of magnitude by taking advantage of the parallelism available in today's computer systems while at the same time achieving a high parallelization efficiency. We intend to address the current limitations and provide an implementation and empirical evaluation in the future.

6. ACKNOWLEDGMENTS

The authors would like to thank Jens Kehne and James McCuller for their valuable input and technical assistance.

7. REFERENCES

[1] L. Albertsson et al. Using complete system simulation for temporal debugging of general purpose operating systems and workloads. MASCOT, 2000.

[2] E. Argollo et al. Cotson: Infrastructure for full system simulation. *SIGOPS*, 43(1), 2009.

[3] P. Barham et al. Xen and the art of virtualization. SOSP. ACM, 2003.

[4] F. Bellard. Qemu: A fast and portable dynamic translator. USENIX, 2005.

[5] P. Bohrer et al. Mambo: A full system simulator for the powerpc architecture. *SIGMETRICS*, 31(4), 2004.

[6] J. Ding et al. Pqemu: A parallel system emulator based on qemu. ICPADS. IEEE, 2011.

[7] G. Dunlap et al. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS*, 36(SI), 2002.

[8] S. Girbal et al. Dist: A simple, reliable and scalable method to significantly reduce processor architecture simulation time. volume 31. ACM, 2003.

[9] A. Jeffery. *Using the LLVM compiler infrastructure for optimised, asynchronous dynamic translation in QEMU*. PhD thesis, University of Adelaide, 2009.

[10] A. Kivity et al. kvm: the linux virtual machine monitor. volume 1. Linux Symposium, 2007.

[11] A. KleinOsowski et al. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1(1), 2002.

[12] R. Lantz. Fast functional simulation with parallel embra. Citeseer, 2008.

[13] K. Lawton. Bochs: A portable pc emulator for unix/x. *Linux Journal*, 1996(29es), 1996.

[14] P. Magnusson et al. Simics: A full system simulation platform. *Computer*, 35(2), 2002.

[15] L. Michel et al. Qemu tcg enhancements for speeding-up the emulation of simd instructions. In *QEMU Users' Forum*, 2011.

[16] J. Miller et al. Graphite: A distributed parallel simulator for multicores. HPCA. IEEE, 2010.

[17] A. Nguyen et al. Accuracy and speed-up of parallel trace-driven architectural simulation. IPPS. IEEE, 1997.

[18] A. Patel et al. Marss: A full system simulator for multicore x86 cpus. DAC. IEEE, 2011.

[19] A. Portero et al. Simulating the future kilo-x86-64 core processors and their infrastructure. Society for Computer Simulation International, 2012.

[20] M. Rosenblum et al. Simos: A fast operating system simulation environment. Technical report, 1994.

[21] M. Rosenblum et al. Using the simos machine simulator to study complex computer systems. *TOMACS*, 7(1), 1997.

[22] M. Sheldon et al. Retrace: Collecting execution trace with virtual machine deterministic replay. 2007.

[23] T. Sherwood et al. Automatically characterizing large scale program behavior. volume 30. ACM, 2002.

[24] M. Sun et al. Fast, lightweight virtual machine checkpointing. 2010.

[25] C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS*, 36(SI), 2002.

[26] K. Wang et al. Parallelization of ibm mambo system simulator in functional modes. *SIGOPS*, 42(1), 2008.

[27] V. Weaver et al. Using dynamic binary instrumentation to generate multi-platform simpoints: Methodology and accuracy. *HiPEAC*, 2008.

[28] E. Witchel et al. Embra: fast and flexible machine simulation. *SIGMETRICS*, 24(1), 1996.

[29] C. Won et al. A detailed performance analysis of udp/ip, tcp/ip, and m-via network protocols using linux/simos. *High Speed Networks*, 13(3), 2004.

[30] M. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. ISPASS. IEEE, 2007.

[31] H. Zeng et al. Mptlsim: A cycle-accurate, full-system simulator for x86-64 multicore architectures with coherent caches. *SIGARCH*, 37(2), 2009.