*Revised version **2017-06-30:** See Revision history.*

Adaptive Cache Warming for Faster Simulations

Gustaf Borgström Andreas Sembrant David Black-Schaffer
Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05, Uppsala, Sweden
{gustaf.borgstrom, andreas.sembrant, david.black-schaffer}@it.uu.se

ABSTRACT

The use of hardware-based virtualization allows modern simulators to very quickly fast-forward between sample points and regions of interest. This dramatically reduces the simulation time compared to traditional functional forwarding. However, as the fast-forwarding takes place through virtualized execution on the native hardware, it is unable to warm simulated structures, such as caches. As a result, sampled simulations taking advantage of virtualization for fast-forwarding find their execution time dominated by functional warming.

To address the cost of warming, we present Adaptive Cache Warming (ACW), a new fast method that determines how much warming each sample/phase/application needs. ACW takes advantage of the virtualization-based fast-forwarding to search for the minimum warming time required during simulation. To determine when the cache is sufficiently warm, ACW uses heuristics based on the last-level cache's cold-set misses.

Our results show that typical practice of conservatively warming last-level caches for around 100M instructions is a vast overkill for nearly all checkpoints. By using ACW, we can adapt the warming per-sample and speedup the simulation by 6.9–18× on average (512× speedup maximum) depending on cache size (2-32MB).

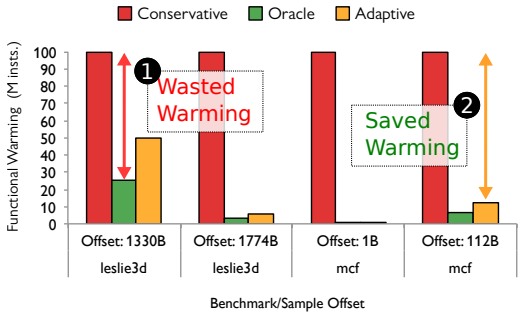

Figure 1: Millions of instructions of functional warming before detailed simulation for 4 samples from 2 applications. The minimum warming for each sample is shown by the Oracle. The Conservative approach uses the same amount of warming for all samples in the benchmark suite. This wastes simulation time by warming the caches more than necessary for most samples (1). Adaptive (ACW) reduces the amount of warming by determine how much warming each sample needs during simulation (2).

1. INTRODUCTION

State-of-the-art computer-system architecture simulators (e.g., Gem5 [3] and MARSS [9]) use hardware-based virtualization to rapidly fast-forward between samples [11] and phases/regions-of-interest [13]. This significantly speedup the simulation since virtualized forwarding can run at near-native processor speed and is therefore much faster than traditional functional forwarding (e.g., 2500× faster on an 2.8 GHz AMD Phenom processor). However, the simulated micro-architectural state belonging to the processor and caches is not updated during virtualized fast-forwarding. The caches and other components must therefore be warmed by a non-virtualized simulation mode before detailed simulation begins. As detailed simulation is often very quick for large numbers of samples [14], the result is that functional warming dominates the simulation time (70-90% of the simulation time in functional warming [11]).

To address the cost of functional warming, we present Adaptive Cache Warming (ACW), a new method that enable us to reduce the amount of time we spend in functional warming mode. Unlike traditional warming, where all samples conservatively use the same amount of cache warming, ACW dynamically adapts the amount of warming for each checkpoint/sample during simulation. The effect of tailoring warming to just what a sample needs is shown in Figure 1 for two applications and samples from SPEC2006 [6]. Traditionally, **Conservative** warming uses a fixed amount of warming for all samples that is set to ensure that the worst sample will produce accurate results (a typical value is 100M instructions of warming). However, this one-size-fits-all approach wastes simulation time since many applications, and even samples within an application, need much less warming than others ❶. To reduce the functional warming overhead, **Adaptive Cache Warming (ACW)** determines how much warming each sample needs during simulation. ACW enable the simulator to dramatically re-

RAPIDO '17 January 23-25, 2017, Stockholm, Sweden

ACM ISBN 978-1-4503-4840-9/17/01.
DOI: http://dx.doi.org/10.1145/3023973.3023974

duce the amount of warming ❷. As a result, ACW speeds up the simulation by 6.9–18× (Adaptive vs. Conservative) depending on cache size (2-32MB).

To determine if the cache is warm enough, we use the optimistic and pessimistic IPC error heuristic proposed by Sandberg et al. [11]. This heuristic treats cache accesses that hit in un-warmed sets (cold-set misses) alternately as hits and misses, and uses this to determine the overall impact on the simulation of the current degree of warming. ACW then increases warming until we detect a sufficiently low IPC error estimate. To speed up the simulation further, ACW can trade-off some accuracy for more speedup. By allowing 0.05 IPC error, ACW can speed up the simulation further from 6.9–18× to 23–56× (2–32MB cache sizes).

Because ACW measures the effect of the current degree of warming and adjusts as needed, it automatically adapts to different conditions, such as cache size, prefetcher, sample, phase, and application. This is especially useful for HW/SW co-design, simulations of JIT compiled programs or memory system research, where storing memory system state with sampled checkpoints is not possible due to the changes on the program binary and/or the underlying simulated system, and where predicting the required warming for each change would be difficult.

We make the following contributions:

- A new method that dramatically reduces the amount of time spend on functional warming for simulators with virtualized fast-forwarding. This speeds up the simulation by 6.9–18×.
- A new method that enables the user to intuitively trade-off some accuracy for faster simulation. By allowing 0.05 IPC error, we can speed up the simulation from 6.9–18× to 23–56×.
- We explore how much warming is needed for different last-level cache sizes (2-32MB), and show that ACW delivers excellent speedups and accuracy across different simulation setups.

2. ADAPTIVE CACHE WARMING

The goal of ACW is to minimize the amount of time the simulator spend doing functional warming. To do so, ACW needs to determine how much warming is needed for each sample during simulation. ACW accomplishes this through an iterative process that starts with a small amount of warming and increases it if our optimistic/pessimistic heuristic indicates too large an error due to cache warming. To do this efficiently, we leverage two techniques: hardware-based virtualization and optimistic/pessimistic error estimates.

Virtualization. Hardware-based virtualization enable us to fast forward at near-native execution speed. This dramatically changes where the simulator spends its time. Figure 2 shows two simulation samples for a sampled simulation (i.e., many short detailed simulation samples spread out over the whole application's execution). For PFSA and SMARTS, most of the instructions are executed in fast-forwarding mode, but since virtualized fast-forwarding runs at near native speed, most of the simulation time is spend on functional warming. Note that while we use the same SMARTS [14]-like approach used in pFSA [11], our adaptive cache warming is applicable to any sample simulation approach that requires warming, such as SimPoint [13], random sampling, etc.

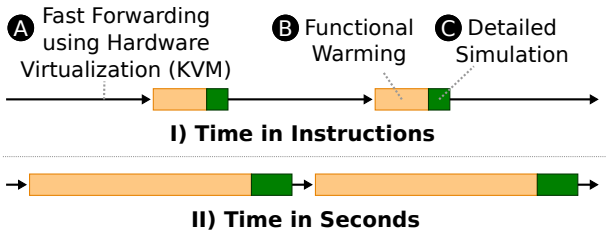

Figure 2: Sampled simulation uses multiple simulation samples [13, 14, 11]. Virtualized fast forwarding (A) can be used to rapidly move between samples without the need to store checkpoints, or when they cannot be stored due to HW/SW co-design. In sampled simulation, caches and other state are warmed using functional warming (B) before detailed simulation is performed to evaluate the sample (C). Although most instructions are executed in fast forwarding mode (I), most of the time is spent in functional warming (II) due to the near-native speed of virtualized fast-forwarding.

We leverage the speed of virtualized fast-forwarding to efficiently jump to different points to adjust the amount of warming. Figure 3 shows an overview of how ACW works. ACW first creates an in-memory checkpoint (implemented with a simple Unix fork) and fast-forwards to the minimum warming period ❶. ACW then warms the caches with functional warming ❷, and then does two detailed simulations to determine the optimistic and pessimistic results for that degree of cache warming ❸. If the error estimate is too large, ACW restarts from the in-memory checkpoint and tries again with increased warming (❹, ❺, ❻). ACW continues until there is sufficient warming that the error estimate is below the target (❼, ❽, ❾). Note that doubling the warming at each step will incur the overhead of all the earlier warmings by the time it finds the appropriate warming.

Error Estimates. To determine if we have enough warming (❸ too short, ❻ too short, ❾ right length), we use the optimistic and pessimistic error heuristic [11]. The idea is to perform two detailed simulations[1] with the same amount of functional warming: one for the pessimistic assumption that any cache miss due to insufficient warming would have been an actual miss with enough warming, and one for the optimistic assumption that misses due to insufficient warming would have been hits with enough warming. From these two simulations we can determine how much the current state of warming could impact the performance of our application.

To determine if a miss due to insufficient warming would have been a hit with enough warming, we track *cold-set misses*. We consider a set to be warm if all of its available cache line slots have been touched (e.g., all eight ways in an 8-way associative cache are used). A cold-set miss is therefore a miss to a set that still has untouched cache line slots. A miss on a cold set may contain the requested data if more warming was used. The optimistic simulation therefore treat the cold-set miss as a hit.

An important observation is that the cache only needs to contain data that is actually used during the detailed

[1]Note that the amount of time spent in detailed simulation is vastly less than that spent warming, so the cost of two detailed simulations is small.

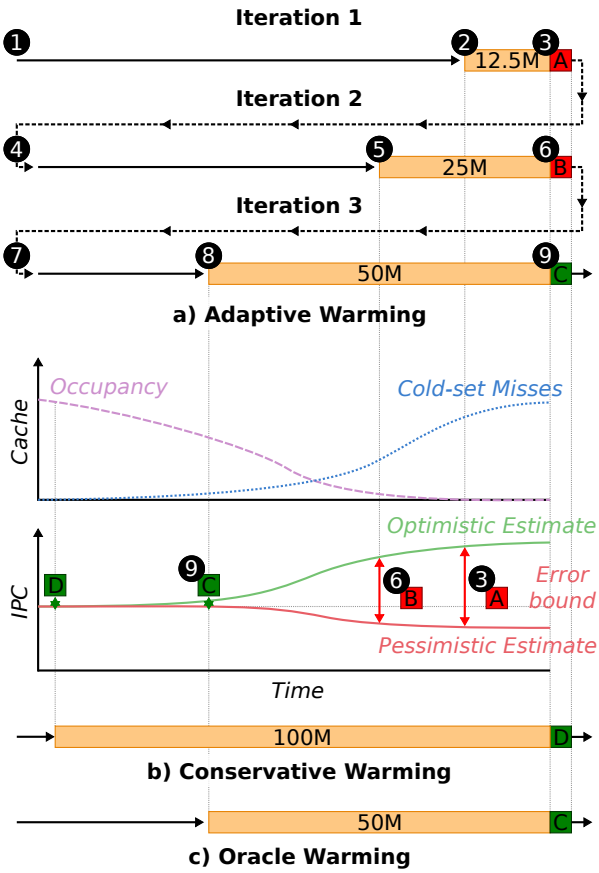

Figure 3: Adaptive Cache Warming is shown for three iterations of increasing cache warming. Virtualized fast-forwarding is used to move from an in-memory checkpoint (1) to 12.5M instructions away from the sample for functional warming (2), followed by detailed warming (3). If the optimistic/pessimistic heuristic indicates too large an error, ACW uses virtualized fast-forwarding to restart from the previous in-memory checkpoint (4), fast forward to 25M instructions before the sample for a longer functional warming period (5) before detailed simulation (6). This process continues until the error estimate is below the target error (7-9). The conservative approach (b) uses the same amount of warming for all samples whereas an Oracle (c) finds the shortest amount of warming without any searching.

simulation: warming for longer may fill other parts of the cache, but unless the sample touches those portions it will not affect the results. That is, cache occupancy is not a good metric to reduce the amount of warming since the cache does not have to be full to be warm enough for accurate simulation. Instead we want a metric that helps us measure whether we have the needed data in the cache.

Moreover, since the optimistic and pessimistic simulations give us an estimate of the maximum warming effect on the error, we can use them to increase simulation speedup by trading-off accuracy for reduced warming.

Summary. ACW uses hardware-based virtualization to cheaply test different functional warming periods, and it uses optimistic and pessimistic estimates of the impact of the current degree of cache warming to determine if more warming is needed.

Frequency	2.5 GHz
Width: F / D / R / I / W / C	8 / 8 / 8 / 8 / 8 / 8
ROB / IQ / LQ / SQ	192 / 64 / 32 / 32
Int. / FP Registers	256 / 256
L1 Instruction / Data Caches	32kB, 64B, 8-way, LRU, 4c
L2 Unified Cache	128kB, 64B, 8-way, LRU, 6c
L3 Shared Cache	2/8/32MB, 64B, 16-way, LRU, 20c
DRAM	SimpleMemory, 3GB, 30ns
ACW Warming Periods	100M, 50M, 25M, 12.5M, 6.2M, 3.1M, 1.6M, 781k, 391k, and 195k.
ACW IPC Target Errors	0.01, 0.02, 0.05 and 0.10

Table 1: Processor configuration.

3. RESULTS

3.1 Methodology

We implemented ACW in the gem5 simulator [3]. Table 1 shows the processor configuration. We evaluate three last-level cache sizes (L3): 2MB, 8MB, and 32MB. To evaluate the method, we use the SPEC2006 [6] benchmark suite. We use ten uniformly distributed checkpoints per application and input. We compare ACW with Conservative (all samples use 100M instructions functional warming since the amount of warming is set very conservatively such that the most demanding sample in the benchmark suite will not lose accuracy) and Oracle (the minimum amount of warming found using ACW without any search overhead). To evaluate the performance, we used simulation time from KVM- and functional simulation-based gem5 runs.

3.2 Gem5 Extensions

To support pessimistic and optimistic error estimates, we had to add functionality to track cold-set misses. To do so, we extend each cache set with a counter that determines how many "cold" cache lines the set contains. The counter is initialized equal to the cache's associativity and then decremented when data is installed. The cache set is thus warm when the counter reaches 0. A cold-set miss is handled differently depending on whether we are doing a pessimistic or optimistic simulation.

- Pessimistic: A cold-set miss is reported as a real miss (i.e., memory latency).
- Optimistic: A cold-set miss is viewed as present in the cache and reported as a *hit* (i.e., L3 hit latency). We implement this as a "prioritized" memory request, that fetches the data immediately from memory without any memory latency.

The output of these two simulations produce an error estimate where the optimistic IPC $\geq$ pessimistic IPC. The true IPC is somewhere in between. In our evaluation, we only consider the last-level cache (L3) since the lower level caches should be warm if the last-level cache is warm.

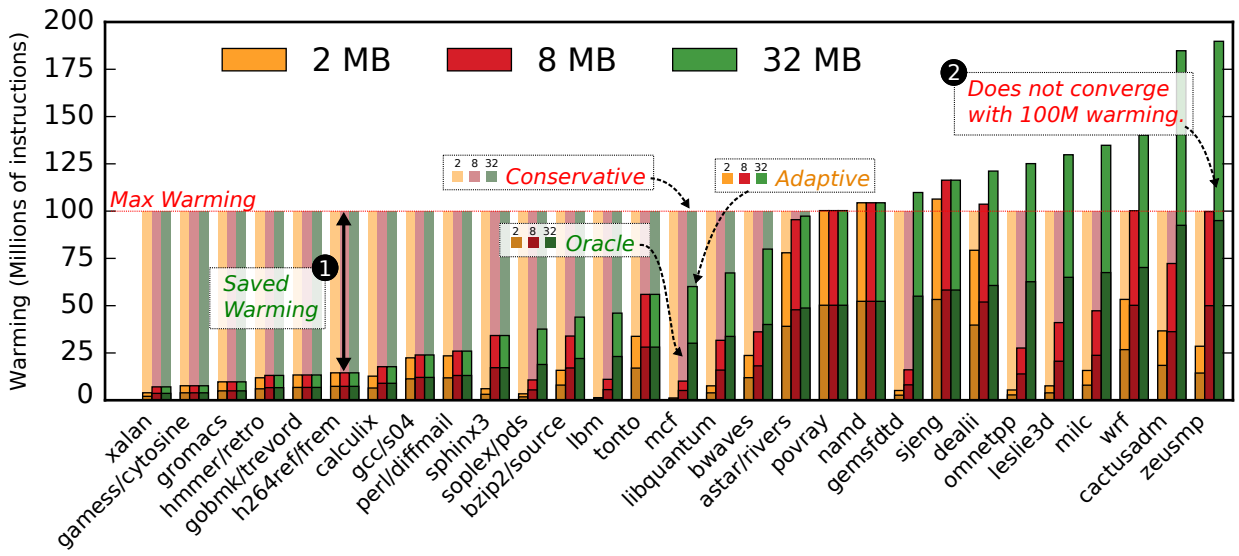

Figure 4: The amount of functional warming for different cache sizes (2, 8, and 32MB). The darker shaded area at the bottom of each bar shows the minimum amount of warming needed without any search (Oracle), the middle shows the total amount of warming ACW needs (search + final warming). The lighter shaded bars behind shows how much Conservative needs to warm the caches based on the worst sample in each application. ACW uses a 0.01 IPC error limit.

3.3 Adaptive Cache Warming

Figure 4 shows the average amount of functional warming used by each application, for three different cache sizes (2, 8, and 32MB), with an maximum error target of a 0.01 IPC deviation from Conservative. The figure also shows, Oracle (dark bars) and Conservative (light bars). Conservative shows the warming needed for the most demanding sample for the given cache sizes.

Overall, ACW adapts the functional warming to the different cache sizes and applications. This saves a significant amount of simulation time compared to Conservative ❶. Across all applications, ACW speeds up the simulation by 18×, 9.7×, and 6.9× for 2, 8, 32MB, respectively. The maximum speedup is 512×, which occurs when ACW and Oracle warm the same minimum amount (i.e., the right/minimum amount of warming is found in the first iteration). The speedup drops slightly with larger caches. This is to be expected since larger caches typically need more warming.

Note that there are some samples that need a lot more than 100M instructions warming. For example, `zeusmp`'s samples are very cache sensitive, as almost none of the optimistic and pessimistic estimates converges when reaching the 100M maximum warming threshold ❷. We investigated this by warming the caches for up to 400M instructions. While almost all cache sensitive applications benefited from this (i.e., the number of samples that did not converge dropped from 94 to 17) there were still a few samples that needed more than 400M instructions to converge. Due to limited simulation time, we could not do "infinite" warming, i.e., where we warm the caches from the start of the application.

3.4 Speedup vs. Accuracy

Figure 5 shows the speedup as a function of simulation error. By tolerating more error, we can reduce the amount of warming further and speedup the simulation more. Note that the speedup includes overhead of the adaptive warming search (i.e., the extra time to find the right amount of warming). The accuracy is defined as the IPC difference between ACW and Conservative. We show the results for 0.01, 0.05, and 0.10 target IPC errors. An IPC error of 0.10 is high, but we include it to illustrate the trade-off between speedup and accuracy.

As expected, the speedup increases as we allow more IPC deviation. For example, (6b: Demanding) moves up from 0× (0.01) to 55× (0.05) to 121× (0.10), (6d: Flexible) moves from 5× (0.01) to 23× (0.05). However, the average error remains relatively low even when we allow more error since only a subset of the applications can take advantage of the speedup vs. accuracy trade-off. For example, (6a: Tolerant) does not move and has a 161× speedup for all error targets.

Most samples fall below the target error ❶. However, in some cases the error is larger than the target error ❷. This happens when the assumption that more warming improves performance does not hold. Some samples benefit from less warming because of the complexity of out-of-order processors. We are still investigating why cache warming is not always a monotonically increasing function, and how ACW can address this issue.

3.5 Cache Warming Behavior

As we have seen in Figures 4 and 5, different applications react differently to the amount of warming they receive. We have observed four different behavior: Tolerant, Demanding, Adaptable, Flexible. These are illustrated in Figure 6 and discussed below.

Tolerant (calculix). These samples are insensitive to cache warming and perform well with very little cache warming ❶. For Tolerant applications, Oracle and ACW use the same amount of warming and will show the highest possible

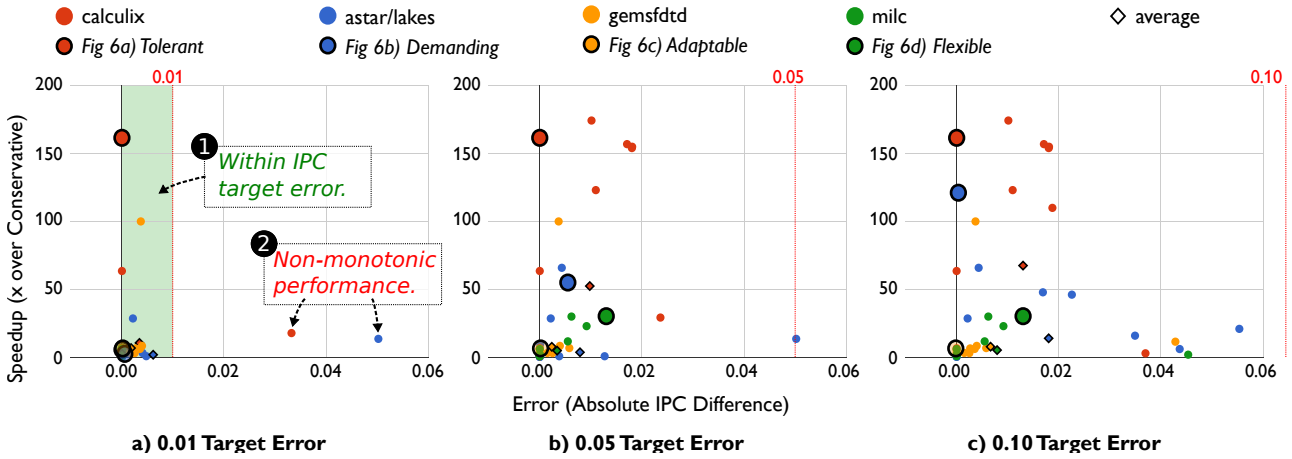

Figure 5: Speedup (including warming period search overhead) vs. Error (IPC difference compared to 100M conservative warming) for an 8MB last-level cache. The dots represent different samples. The big black lined circles highlights the samples that we look in more detail in Figure 6 (next figure).

Target IPC Error	Real IPC Error (Avg.)			Speedup (Geomean)		
0.01	0.007	0.012	0.012	18×	9.8×	6.9×
0.02	0.009	0.013	0.013	25×	14×	10×
0.05	0.010	0.017	0.017	56×	33×	23×
	2MB	*8MB*	*32MB*	*2MB*	*8MB*	*32MB*

Table 2: Speedup vs. Target IPC Error for 2, 8, 32MB cache sizes. Avg. shows the geometric mean across all sample points.

speedup over Conservative since they always use the minimum amount of warming.

Demanding (astar/lakes). In contrast, demanding samples need a lot of warming. For example, `astar/lakes` needs the full 100M warming ❷. Here, all three, Oracle, ACW and Conservative use the same amount of warming.

Adaptable (gemsfdtd). These samples have very different behavior depending on how much warming they receive. Too little warming and the IPC error is too large ❸, but very small error if they receive a little bit more warming ❹. Here, ACW is effective at finding the minimum amount of warming the application can tolerate.

Flexible (milc). The speedup changes gradually as a function of warming. These samples are good candidates for trading off some accuracy for more speedup. For example, if we allow only 0.01 IPC error (basically no error), ACW can reduce the amount of warming from 100M to 12M ❺. However, if we allow a small 0.05 IPC error, ACW can reduce the amount of warming further down to only 781k instructions ❻.

3.6 Summary

The results show that ACW significantly reduces the amount of warming, and it can improve the performance further by trading off some accuracy for more speedup. Table 2 summarizes the results. On average, ACW speeds up the simulation by 18×, 9.7×, 6.9× for an 2, 8, 32MB caches, respectively, and by 56×, 33×, 23× if we allow a 0.05 IPC error.

4. RELATED WORK

Sampled Simulation. Sampled simulation is used extensively to reduce simulation time, e.g., SMARTS [14], or by the selection of representative samples, e.g., SimPoint needed [12, 13, 10]. The value of knowing which part of an application to simulate has even made its way into benchmarks, such as with PARSEC benchmark [2], where the applications indicate the regions of interest themselves.

In all cases, sampled simulation requires that the simulation state is warm before collecting detailed statistics.

Warmup Methodologies. There have been many works on identifying the correct amount of warming. BLRL [4] counted reuse latencies in a simulation trace to statistically determine the amount of warming needed for each sample. This was extended by Lou et. al. [8] by keeping track of "cold-start accesses". This latter approach is similar to Sandberg's "cold-set misses", however it lacks the optimistic/pessimistic information which allows us to determine when to stop warming.

While these methods can accurately determine how much warming is needed, they assume an always-on warming methodology. This is helpful for determining a conservative warming bound for checkpoints which will be re-simulated many times if you cannot save the processor and cache state. Our approach assumes that we do not have always-on warming and need to dynamically adjust the warming, but that we do have very fast virtualized fast-forwarding, which allows us to cheaply simulate at different points in the execution.

Virtualization Use in Simulations. Throughout this paper, we have frequently used ideas presented by Sandberg et al [11] to leverage virtualization for fast-forwarding between samples and to find the right amount of warming. Virtualization has also been used by MARSS [9, 15] as well as in TQSIM [7] through QEMU [1] to execute applications and generate execution- and memory traces, whose performance can then be estimated. Recently, Hassani et all demonstrated LiveSim [5], a framework that uses virtualization to store processor and system checkpoints in memory for rapid re-simulation of the same checkpoints.

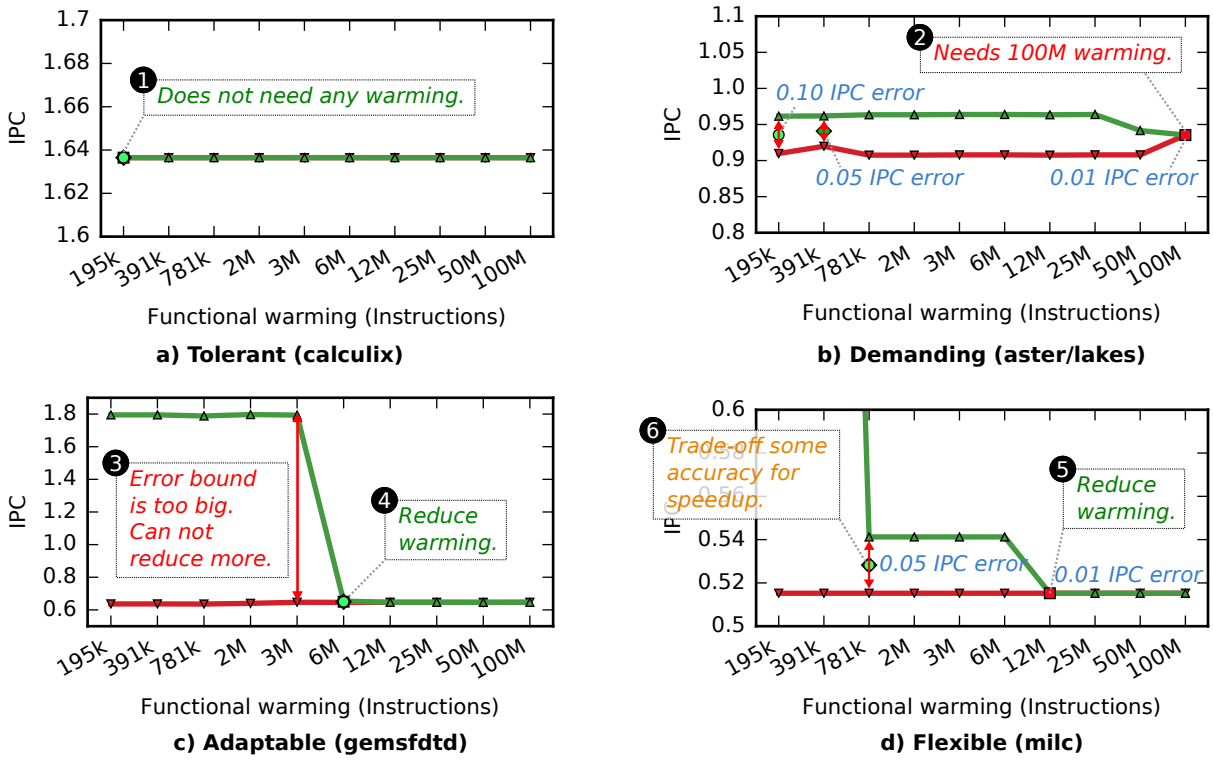

Figure 6: Applications react differently to the amount of warming they receive. We have identified four types of behavior: a) Insensitive, b) Sensitive, c) Adaptable, d) Flexible.

5. CONCLUSIONS

Today's computer architectural simulators spend most of their time doing functional warming. In this paper, we presented Adaptive Cache Warming (ACW). ACW determines during simulation time how much cache warming is needed for each simulation sample. It leverages hardware-based virtualization and optimistic/pessimistic error bounds to find the right amount of warming quickly. To obtain even greater speedups, ACW allows the user to trade accuracy for even faster simulation. Our results show that ACW speedups up the simulation by 6.9–18× on average (and 23–56× by trading off accuracy with a 0.05 IPC error) depending one cache size (2-32MB).

6. ACKNOWLEDGMENTS

This work was funded in part by the Swedish Science Council (grant 2014-5480), the Swedish Foundation for Strategic Research (grant FFL12-0051), the Uppsala Programming for Multicore Architectures Research Center, and the Swedish National Infrastructure for Computing (SNIC) at UPPMAX.

7. REFERENCES

[1] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 Simulator. *SIGARCH Comput. Archit. News*, 2011.

[4] L. Eeckhout, Y. Luo, K. D. Bosschere, and L. K. John. BLRL: Accurate and Efficient Warmup for Sampled Processor Simulation. *The Computer Journal*, 48(4):451–459, Jan. 2005.

[5] S. Hassani, G. Southern, and J. Renau. LiveSim: Going Live with Microarchitecture Simulation. In *Proc. International Symposium on High-Performance Computer Architecture (HPCA)*, 2016.

[6] J. L. Henning. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News*, 2006.

[7] S.-h. Kang, D. Yoo, and S. Ha. TQSIM: A fast cycle-approximate processor simulator based on QEMU. *Journal of Systems Architecture*, 66âĂŞ67:33–47, May 2016.

[8] 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.

[9] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSS: A Full System Simulator for Multicore x86 CPUs. In *Proc. Design Automation Conference (DAC)*, 2011.

[10] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for Accurate and Efficient Simulation. In *Proc.*

International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), 2003.

[11] A. Sandberg, N. Nikoleris, T. E. Carlson, E. Hagersten, S. Kaxiras, and D. Black-Schaffer. Full Speed Ahead: Detailed Architectural Simulation at Near-Native Speed. In *Proc. International Symposium on Workload Characterization (IISWC)*, 2015.

[12] T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[13] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proc. Internationl Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[14] 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)*, June 2003.

[15] M. T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *Proc. International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2007.

Target IPC Error	Real IPC Error (Avg.)			Speedup (Avg.)		
0.01	0.007	0.012	0.012	103×	93×	92×
0.02	0.009	0.013	0.013	124×	112×	110×
0.05	0.010	0.017	0.017	199×	185×	183×
	2MB	*8MB*	*32MB*	*2MB*	*8MB*	*32MB*

Table 3: Speedup vs. Target IPC Error for 2, 8, 32MB cache sizes. Avg. shows the arithmetic mean across all sample points.

Revision history

2017-06-30: In the paper, we previously used the *arithmetic* mean for average speedup. However, the arithmetic mean produce overly optimistic performance numbers since it does not weight speedup and slowdown equally. In this revision, all speedup averages have been changed to show geometric mean instead. The "average" points in Figure 5 have also been changed accordingly over the "Speedup" axis.

Table 3 shows the arithmetic mean of the speedups over all sample points. The geometric mean in Table 2 in Section 3.4 is lower in comparison, but the results show that the simulation technique still produces significant performance improvements. The conclusions remain the same.