Simulator Calibration for Accelerator-Rich Architecture Studies

Mochamad Asri[1], Ardavan Pedram[2], Lizy K. John[1], and Andreas Gerstlauer[1]
[1] The University of Texas at Austin, Austin, TX, USA
[2] Stanford University, Stanford, CA, USA

Abstract—**Cycle-accurate simulators are widely used in architecture research. A certain degree of performance inaccuracy is expected in such simulators. Such mismatches are tolerable in micro-architectural studies that emphasize relative performance. However, when modeling accelerator-rich heterogeneous systems, an inaccurate baseline CPU model can lead to severe over/under-estimation of the speedup and/or energy savings. In this paper, we present a systematic approach and methodology for calibration of cycle-accurate simulators targeting such studies. We further compare the potential impact that an unrepresentative baseline CPU model can have on heterogeneous system design and describe how it could cause to misleading design evaluations.**

We demonstrate our approach on calibration of MARSSx86, a widely-used, cycle-accurate x86 system simulator. Using our methodology, we calibrate MARSSx86 to closely match the performance of state-of-the-art Intel machines targeting high-performance computing (HPC) benchmarks. Our calibrated MARSSx86 shows on average less than 10% error across a wide range of HPC and general-purpose benchmarks. Using this calibrated baseline simulator, we further quantify the impact of such calibration on an accelerator-rich architecture case study. Our results show that an accelerator estimated to obtain a speedup of 10.3x using unrefined simulators can in reality only achieve a 5.3x speedup when an accurate CPU model is employed.

I. Introduction

Heterogeneous systems have emerged as state-of-the-art computing solutions, providing massive compute capabilities within limited power budgets. Such systems increasingly integrate accelerators together with traditional host processors at various levels of task granularity, programming model, and memory hierarchy [8]. Different system constraints and requirements, such as typical task sizes, and the data communication patterns between the host and the memory will drive the selection of design options. The potential speedup and energy savings are direct results of such design choices. The overheads incurred by integrating accelerators need to be amortized by the energy savings and speedup achieved by offloading computations to a faster and more efficient accelerator. Architecture researchers need fast and accurate methods to validate such savings and be able to tune their accelerator architecture and integration options.

In order to perform such studies, researchers rely heavily on cycle-accurate simulators. However, as pointed out in previous work [9], [12], a certain degree of experimental error and performance mismatch are unavoidable. Some of the mismatches are the result of modeling the system at a lower level of detail compared to RTL or gate-level simulation, thereby trading accuracy for speed. This can cause a significant discrepancy between the model and the targeted system. Such mismatches may be tolerable in micro-architectural studies where the emphasis is often on the relative but not absolute effect of architectural modifications on performance. Nevertheless, an inaccurate baseline model can still lead to wrong conclusions, e.g. when relative contributions of different micro-architecture components are not properly accounted for. Moreover, an accurate baseline is essential when modeling heterogeneous systems. An inaccurate baseline CPU model can lead to severe over- or under-estimation of the speedups and energy savings achieved by integrating accelerators. It is therefore crucial that simulators used in both system- or micro-architecture studies be calibrated to match real-world platforms.

In this work, we demonstrate a simulator calibration methodology and evaluate the potential impact such a calibrated simulator can have on accelerator-rich architecture studies. Using microbenchmarking techniques, we show a systematic, step-by-step approach to calibrate a widely-used cycle-accurate x86 simulator, starting from validation of core pipeline operation up to memory system behavior. We target high-performance computing (HPC) applications built around core dense linear algebra operations as a domain in which accelerators are extensively used. We use MARSSx86 [16] as the baseline simulator for this work. We chose MARSSx86 since it is the full system simulator with the broadest x86 support. Specifically, MARSSx86 supports the widest range of vectorized ISA extensions among all cycle-accurate simulators. This is essential in providing a realistic baseline for HPC applications that leverage such vector extensions for improved floating-point performance.

The rest of the paper is organized as follows: Section 2 provides a discussion on related work. Section 3 describes our calibration methodology. Section 4 presents evaluation of the calibrated simulator and the potential impact on heterogeneous system studies. Finally, Section 5 concludes the paper with a summary and outlook on future work.

II. Related Work

In the following, we first discuss related work in existing system simulators, including the rationale driving the selection of the baseline simulator used for our studies. We further present prior art in heterogeneous system research utilizing such full-system simulators, including key differences of our work from prior art.

Cycle-Accurate Simulators. One of the earliest simulators in the architecture community is SimpleScalar [5]. Since then, many simulators have emerged and are able to model complete computer systems including both user and operating system (OS) code, such as Zsim [18], Sniper [7], gem5 [4] and MARSSx86 [16].

Zsim [18] and Sniper [7] use binary instrumentation or interval simulation to provide fast and scalable system validation. They achieve high simulation speeds by trading off accuracy and raising the level of abstraction in modeling of architectural features, especially for hardware and OS interactions. However, as our results will show, baseline accuracy and architectural detail of a simulator are essential for modeling of heterogeneous systems, where overheads of interactions between accelerators, the OS and the host CPU can have a large influence on overall results. Higher-level simulators such as Zsim and Sniper are unable to support this level of modeling granularity.

Among cycle-accurate full-system simulators, gem5 [4] is a discrete-event simulator supporting x86, ARM, PowerPC, SPARC, and MIPS targets for both in-order and out-of order CPUs. MARSSx86 [16] was designed to provide fast simulation of x86 systems by integrating a x86 timing model with QEMU. As mentioned before, we target HPC applications that rely heavily on vectorized instructions and floating-point performance. In this domain, the potential for acceleration is best exploited. gem5 has only marginal support for vectorized instructions. A number of Streaming SIMD Extensions (SSE) are not completely implemented. This results in many of our targeted HPC benchmarks failing to run on gem5 due to unimplemented instructions. By contrast, MARSSx86 has a more complete and accurate x86 ISA implementation, including SSE and x87 floating-point support. As such, we use MARSSx86 as the baseline simulator in this work.

Heterogeneous Architecture Studies. Several heterogeneous system studies have employed MARSSx86 as their baseline model to show the potential of their proposed accelerator and its improvements. Esmailzadeh *et al.* [10] proposed a Neural Processing Unit (NPU) tightly coupled to the processor pipeline to accelerate small code regions. Zhu *et al.* [21] proposed a software-assisted hardware accelerator for the critical style-resolution kernel within a Web browser engine. Yang *et al.* [20] presented an approach to utilize the CPU and GPU resources that are integrated on the same die sharing the on-chip L3 cache. While all of the aforementioned heterogeneous system studies show significant speedup benefits, none of them evaluated that their baseline simulator models accurately represent the targeted machine's performance.

Several studies have evaluated the performance mismatch between cycle-accurate full-system simulators and real machines. Butko et al. [6] have investigated the runtime mismatch between a cycle-accurate gem5 model and a real ARM. Gutierrez et al. [12] further leveraged the insights of this work to account for the accuracy of micro-architectural characteristics and how they affect the overall runtime accuracy. While there have been several such studies on identifying sources of errors on a particular cycle-accurate simulator, to the best of our knowledge, there exists no prior work that specifically reports and discusses x86 simulator and MARSSx86 accuracy. Furthermore, there are no published studies that consider the calibration and impact of the baseline simulator accuracy in the context of heterogeneous systems design. Different from previous work, this paper proposes a detailed and systematic x86 simulator calibration methodology that is applied to obtain a highly accurate MARSSx86 model. Moreover, we quantify the implication of such calibrations on a heterogeneous system design and design space exploration case study.

TABLE I
SYSTEM CONFIGURATION.

Parameter	i7-920	MARSSx86 Model
Fetch Width	4	4
Dispatch Width	4	4
Issue Width	5	5
Commit Width	4	4
Writeback Width	4	4
Cache Block Size	64B	64B
L1 I-cache Size	128kB	128kB
L1 I-cache Associativity	8-way	8-way
L1 D-cache Size	128kB	128kB
L1 D-cache Associativity	8-way	8-way
L2 Cache Size	256kB	256kB
L2 Associativity	8-way	8-way
L3 Cache Size	8MB	8MB
L3 Associativity	16-way	16-way

III. Calibration Methodology

In the following, we present our methodology to calibrate a x86 CPU model for HPC applications. We use a bottom-up methodology that starts from the simplest components of a system and gradually treats and evaluates those as pieces within a larger system. This includes gradually developing both the simulated hardware layers as well as the software benchmark stacks on top. The calibration process is partitioned into two main steps: It starts from the core pipeline evaluating its accuracy with the simplest piece of microbenchmark code. Afterwards, we build on top of the calibrated pipeline model to further observe memory system behaviour. As we will describe in the following, we use microbenchmarks of increasing complexity to calibrate the pipeline and the cache model step by step. Using microbenchmarks that go hand in hand with the hardware layer being calibrated helps us to efficiently unravel and tune the unknown detailed micro-architectural parameters of the targeted machine.

While our general step-by-step methodology should be applicable to any x86 simulator or target machine, as pointed out in Section 2, in this paper we specifically demonstrate our calibration methodology as applied to MARSSx86 targeting an Intel Core i7-920 CPU for HPC applications.

We use General Matrix Matrix Multiplication (GEMM) as the basis for our microbenchmarking stack. GEMM is a key kernel at the core of many HPC applications. It is nicely

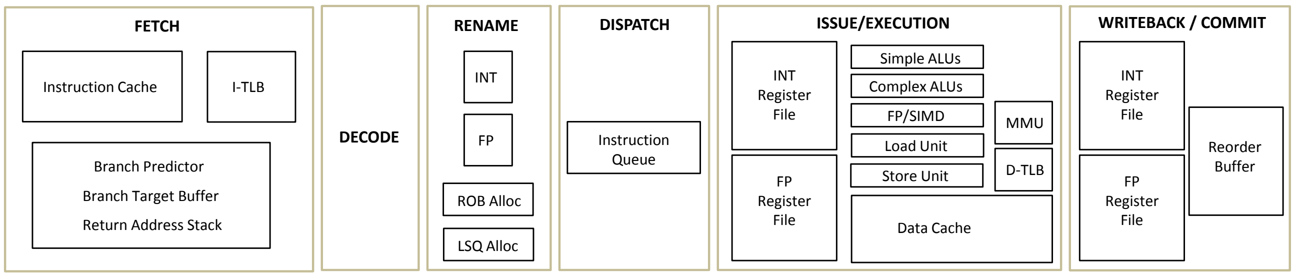


Figure 1. Core Pipeline Architecture.

layered, reflects the peak performance, uses a wide variety of compute-intensive instructions, including vectorized instructions, can exploit locality at all levels of memory hierarchy, and has optimized implementations available [19].

Table I details the MARSSx86 configurations compared to our target Intel processor. We used publicly available information to match the target machine configuration, while simultaneously conducting validation checks on the given configuration through microbenchmarking and calibration. We used the PAPI [15] performance counter tool to get the total number of cycles on the i7-920 as compared to MARSSx86's statistics from its simulated CPU model.

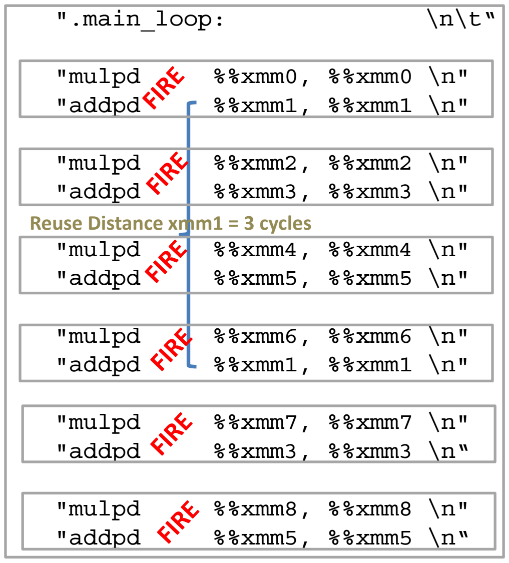
A. Core Pipeline Calibration

A typical out-of-order (OoO) pipeline architecture model is depicted in Figure 1. The front-end of the pipeline (from the FETCH until the DISPATCH stage) works in-order until the RENAME stage. The RENAME stage renames (i.e. maps) the architectural source and destination register IDs into physical Register File IDs and insert the instructions into the Issue Queue (also referred to as Reservation Station).

Instructions wait in the Issue Queue until they are selected for execution. When an instruction is selected, it reads its dependent instructions via the wake-up logic, reads its source operands from the physical Register File, and is executed in Functional Unit(s). After execution, an instruction's result is broadcasted via a bypass network such that any dependent instruction can use it immediately. The instruction retires once it reaches the head of the Reorder Buffer (ROB), and updates the corresponding Register Alias Table.

In order to calibrate pipeline performance, we first use a variant of the inner-most micro-kernel of GEMM that performs a *rank-1 update* [19]. The microbenchmark is designed to evaluate the peak performance of the pipeline. It is modified to exclusively perform register-to-register computations, i.e. without any load/store operations from L1 to registers. This allows us to investigate the execution behavior of the OoO pipeline without any potential interference of the load/store unit and cache system. Using such an instruction stream, we compare the peak performance produced by the pipeline model and the targeted i7 machine. Any discrepancies are a sign of potential issues in the pipeline model.

Figure 2 illustrates our microbenchmark that repeatedly executes SSE multiply and add (*mulpd* and *addpd*) instructions


```
".main_loop:                 \n\t"

"mulpd    %%xmm0, %%xmm0  \n"
"addpd    %%xmm1, %%xmm1  \n"

"mulpd    %%xmm2, %%xmm2  \n"
"addpd    %%xmm3, %%xmm3  \n"

"mulpd    %%xmm4, %%xmm4  \n"
"addpd    %%xmm5, %%xmm5  \n"

"mulpd    %%xmm6, %%xmm6  \n"
"addpd    %%xmm1, %%xmm1  \n"

"mulpd    %%xmm7, %%xmm7  \n"
"addpd    %%xmm3, %%xmm3  \n"

"mulpd    %%xmm8, %%xmm8  \n"
"addpd    %%xmm5, %%xmm5  \n"
```

Figure 2. Microbenchmark for testing instruction latencies.

in a loop, similar to a subset of rank-1 update operations in GEMM. SSE instructions are intensively used in our targeted HPC applications. Each *addpd* and *mulpd* operates on *xmm* registers that each hold 128-bit values and can operate on two double-precision values in a cycle. Furthermore, the i7-920 can perform four double precision (DP) FLOPs/cycle [1]. Consequently, the microbenchmark is expected to run at near-peak performance of four DP-FLOPs/cycle (FPC).

1) Instruction Latencies: At the core level, instruction latencies are one of the first parameters to calibrate against the targeted machine. Any mismatch will directly impact the accuracy of the processor model.

One of the biggest challenges in calibrating instruction latencies is observability. In the simulator, it is trivial to measure the cycles an instruction takes to complete as the behaviour of the model is fully visible and tractable. However, in a real CPU, there is no easy way to measure how many cycles an instruction requires to finish. Hence, a systematic approach to make instruction latencies observable is required.

We explain our approach using Figure 2 as a motivating example. We define the reuse distance of destination registers as the number of distinct references between two references to the same location. Therefore, the reuse distances of *addpd* and *mulpd* instructions are 3 and 6 cycles. Consequently, if the corresponding latencies for *addpd* and *mulpd* are less than 3 or 6, respectively, the microbenchmark will execute in nearly 4 FPC (i.e, peak performance in the target machine). Otherwise, *addpd* and *mulpd* would have to stall and wait for the dependency to resolve, resulting in a degradation in performance. We measured the microbenchmark shown in Figure 2 to execute nearly in peak performance on the i7-920, achieving a FPC of 3.94. We then varied the reuse distances in the microbenchmark until performance degrades to further determine instruction latencies. Using microbenchmarks with varying reuse distances, we measured *addpd* and *mulpd* latencies to be 3 and 5 cycles, respectively.

By contrast, the MARSSx86 model, with a similar configuration as the i7-920, could only achieve an FPC of 2.4. Per-cycle event and log traces reveal that the MARSSx86 pipeline is not able to commit at a rate of 4 FPC because an *addpd* instruction was waiting on the previous *addpd* instructions to retire. We further observed that the instruction latencies of *addpd* and *mulpd* were hardcoded to 6 cycles in MARSSx86. We adjusted latencies to their targeted values of 3 and 5 cycles, respectively. We utilized a similar approach to calibrate the instruction latencies of other SSE instructions. We first determined potential latency candidates, based on the technical documentations provided in [11]. We then crafted similar microbenchmarks with varying reuse distances for these instructions to measure latencies on the target machine and adjust the MARSSx86 model accordingly.

As described in the above example, the key points in calibrating instruction latencies are as follows: First, coming up with a baseline microbenchmark that would run in peak performance in the pipeline. Second, using this baseline, calibrating most frequent arithmetic instructions latencies by including those instructions in the microbenchmark. Finally, one can determine the actual instruction latencies in the targeted machine by modifying the reuse distances in the microbenchmark until the performance deviates from the peak. By employing this generic methodology, one can easily come up with microbenchmark variants in order to determine instruction latencies for any given x86 instruction set.

2) Pipeline Width: Once instruction latencies are calibrated, we move to a higher level in the pipeline hierarchy. In a pipelined system, every stage should ideally have a uniform production and consumption rate. Any non-uniform rate makes the system operate at less than peak performance. For example, in the pipeline architecture depicted in Figure 1, if the processor FETCHes up to four instructions, but can only DECODE two instructions at a time, the system throughput will be limited to two. In this scenario, the DECODE stage is the performance bottleneck.

The pipeline width varies from processors to processor between 2-wide and 8-wide. Hence, validation of the system width is an integral part of the pipeline calibration. In order to determine the system width, we use the following approach: In general, pipeline performance increases as a function of system width, assuming a sufficiently independent instruction stream and no cache miss penalties. Using such an instruction stream, one can vary the system width gradually and observe how performance improves. Note that the system width we refer to here is not merely the issue width. Performance bottlenecks can equally reside in other stages. As such, by width we refer here to the parameters of the entire system, including front-end and back-end. Hence, changing the width to four means that all fetch, rename, dispatch, issue, commit, and writeback widths are set to four accordingly. With this, the pipeline performance should increase as the system width grows.

TABLE II
GEMM MICRO-OP BREAKDOWN COMPARISON.

	i7-920	**MARSSx86**	**Gap**
# of Instructions	229 Million	225 Million	2%
# Micro-Ops	**229.2 Million**	**465 Million**	**202%**

We run the entire GEMM benchmark to increase test program complexity and evaluate the pipeline width across a mix of instructions each progressing differently through different stages of the pipeline. The GEMM benchmark employs a combination of a broad range of compute-intensive and vectorized instructions. We make minor modifications of the benchmark such that every load and store related instruction is omitted, thus avoiding interference of the cache system. The real i7 achieves a performance of 3.94 FPC for this benchmark. By contrast, we observed that the latency-calibrated MARSSx86 could only reach an FPC of 2.45. In addition, increasing the pipeline width of MARSSx86 did not bring expected performance improvements. In our special case, the problem was due to MARSSx86 internally hardcoding the issue width to four. After removing the hardcoded assignment and making issue width configurable based on the provided config file, this problem was fixed. We then observed that with a pipeline width of seven or eight, the MARSSx86 core model produced 90% and 102% of i7-920 peak performance, respectively. Therefore, a pipeline width of seven or eight could be assumed to give a reasonable configuration assuming there is no cache miss penalty. However, this is not truly the case. For example, for variants of the original microbenchmark with large enough reuse distance, a width of seven would artificially inflate performance beyond the peak FPC of 4 that the real machine is capable of. In the following, we explain the root issue and the key insights that support it. The main reason for this mismatch is because in MARSSx86, the system width refers to a granularity of micro-operations (micro-ops) while a a real x86 machine defines widths at instruction granularity. Hence, if we apply a similar width configuration as in the i7, MARSSx86 will achieve a lower number of instructions per cycle (IPC) in cases where it has to execute more micro-ops than instructions compared to the real x86 machine.

3) Instructions and Micro-ops: Table II illustrates the breakdown of instructions and micro-ops of GEMM executing on the i7-920 and MARSSx86. As expected, the number of dynamically executed instructions matches closely within the range of possible measurement errors. By contrast, an x86 instruction is internally broken down into simple RISC-like micro-operations, where the gap between micro-ops executed in the i7-920 and MARSSx86 is 202%. We further examine

TABLE III
MICRO-OPS OF GEMM ON MARSSX86.

SSE		**NON-SSE**	
96%		4%	
Multi-Op	Single-Op	Multi-Op	Single-Op
94%	6%	87%	13%

the detailed micro-op breakdown. Table III presents the breakdown on MARSSx86, where 96% of the micro-ops belong to SSE instructions. Moreover, 94% of the SSE micro-ops belong to multi-op instructions, which are typically broken down into two smaller micro-ops. From this observation, and from data in Table II, we infer that the i7-920 has more single-op SSE instructions than the MARSSx86 core model.

In order to account for this discrepancy, we considered the following options:

1) Altering the internal micro-op implementation in the MARSSx86 core model, which arguably is the ideal option. However, there is no publicly available documentation on how each x86 instruction is broken down into micro-operations.
2) Fixing the processor pipeline width statically to seven or eight to achieve a reasonable performance match for the larger GEMM kernel. However, opting for this approach would make the system configuration of the core model artificially unaligned to the system configuration of the real machine. If different applications have different single- to multi-op instruction ratios, the core model would not correctly capture the behavior.
3) Keeping the base system configuration aligned with the real target machine parameters, but *dynamically* adjusting the *execution width* to account for multi-op instructions. In this way, the adjustment in the core model will be orthogonal to the system configuration.

We followed the third approach and modified the commit logic flow in the MARSSx86 model to support it. Other front-end units such as fetch, rename and issue logic are also modified using the same principle.

Figure 3 shows the flowchart of the improved commit logic. Our approach *dynamically* treats multi-op SSE instructions as single-op instructions at runtime. In the original version shown on the left, the commit logic first checks whether the CPU has already committed more micro-ops than the machines' commit width supports. If yes, then we return. If not, then the logic checks the next entry in the ROB. If a micro-op is not ready to retire, then the logic returns. Otherwise, we commit and increase the commit counter.

As shown in Figure 3(b), we change the commit logic such that when a micro-op is ready to retire, we check whether it belongs to the SSE class. If not, we normally commit and increment the commit counter. However, if it is of SSE type, we first check whether it is a multi-op SSE instruction. Every micro-op in the MARSSx86 model will have *uop.som* and *uop.eom* flags, indicating the first and last operation in a sequence of micro-ops for one instruction. If an instruction is a single-op instruction, both flags will be set to 1. We alter

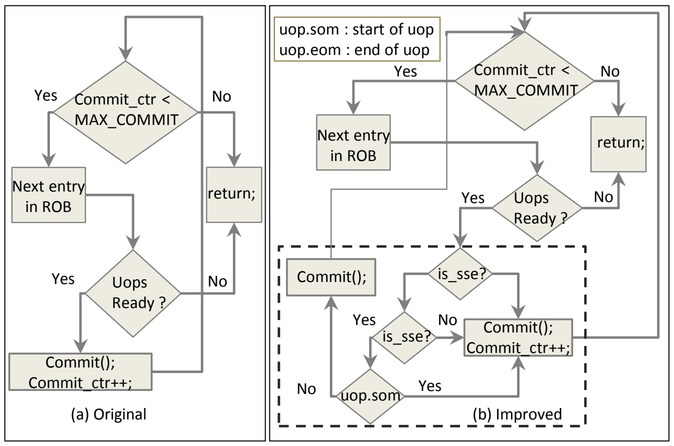


Figure 3. Flowchart of the improved commit logic, accounting for different micro-op implementation in the simulator.

the commit logic flow as follows: If *uop.som* is 1, i.e. it is either the first micro-op of a multi-op instruction or a single-op instruction, we commit and also increment the commit counter. In all other cases, we commit but do not increment the counter. In this way, a multi-op SSE instruction is treated as single micro-operation at commit time. With this, we are able to closely match performance of the real machine for the larger GEMM kernel while keeping the basic pipeline width set at its correct value of four.

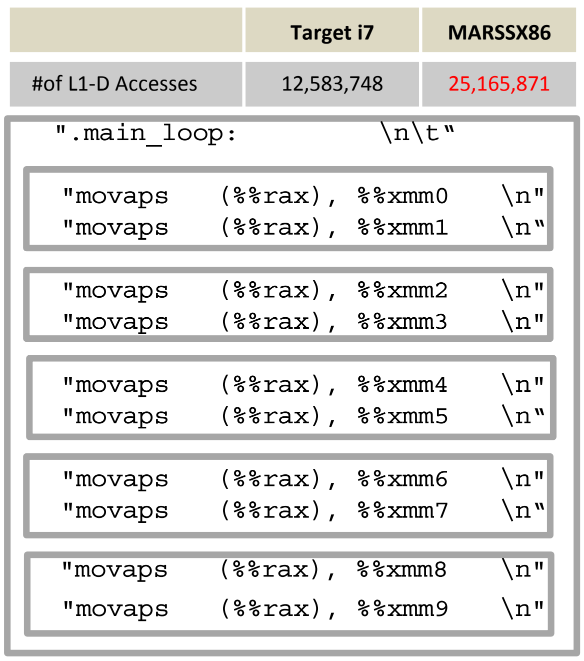
B. Cache System Calibration

In general, there are multiple levels of cache hierarchy in modern processors. A unit validation at each level is required before calibrating the complete cache system.

Our targeted machine has three levels of caches. We examine L1 cache access behaviour first. We run the GEMM benchmark now including load and store instructions. We collected statistics of the number of L1 accesses both from MARSSx86 and the targeted machine. The resulting L1-D statistics show discrepancies. In particular, the number of accesses from MARSSx86 is 74 million, while the targeted machine issued only 46 million L1-D accesses.

The previous experiment gave us a key insight that every SSE instruction is broken down into two micro-ops in MARSSx86. In our previous microbenchmark, we used *addpd* and *mulpd* as a motivating example. In reality, the GEMM benchmark uses a wide variety of SSE instructions, including load-store type of instructions. For such instructions, one load is executed as two load micro-ops. As such, accessing the L1-D cache with a SSE load instruction would logically generates two accesses to the cache.

To confirm this, we crafted a simple microbenchmark similar to previous ones but consisting of SSE load instructions that move data from the cache to *xmm* registers. Figure 4 shows the microbenchmark and the corresponding L1-D accesses when executed in MARSSx86 model and the target machine. As shown in the figure, the number of L1-D accesses in MARSSx86 are twice as high as in the target machine. This confirms that accessing the L1-D cache with a SSE load instruction creates two different accesses in MARSSx86,



	Target i7	MARSSX86
#of L1-D Accesses	12,583,748	25,165,871

```
".main_loop:           \n\t"
  "movaps   (%%rax), %%xmm0    \n"
  "movaps   (%%rax), %%xmm1    \n"
  "movaps   (%%rax), %%xmm2    \n"
  "movaps   (%%rax), %%xmm3    \n"
  "movaps   (%%rax), %%xmm4    \n"
  "movaps   (%%rax), %%xmm5    \n"
  "movaps   (%%rax), %%xmm6    \n"
  "movaps   (%%rax), %%xmm7    \n"
  "movaps   (%%rax), %%xmm8    \n"
  "movaps   (%%rax), %%xmm9    \n"
```

Figure 4. Microbenchmark for testing L1-D accesses.

TABLE IV
CACHE MISS-RATES ON EACH LEVEL.

	MARSSx86			i7-920		
	L1-D	L2	L3	L1-D	L2	L3
Total Accesses	43.3M	4.7M	4M	46.2M	4.8M	207K
Miss Rate	11%	85%	3%	10%	4.5%	11%

TABLE V
L2 SET BREAKDOWN WHEN EXECUTING GEMM.

Set	# of way	# of Competing Blocks	Miss	Hit
# 0	8	19	18,387	2,291
# 1	8	46	15,637	882
....	...	...	...	...
# 56	8	42	15,754	921

whereas it issues only one access in the target machine. We investigated other variants of SSE load and store instructions and found similar mismatches in the number of L1-D accesses.

While this does not affect simulated performance, we made the following changes in the MARSSx86 model to address reported statistics: Whenever encountering a load/store instruction in the pipeline, we first identify whether it is SSE, and then adjust the runtime statistics counter. Whenever there is a cache accesses from an SSE instruction, the cache access will be treated only once. Hence, the statistic collections are aligned with the target machine.

We use the calibrated L1 cache model to further investigate other levels of the cache system. Table IV shows the statistics of cache miss-rates at each level in MARSSx86. Miss-rates are measured when executing GEMM. We observe that the L2 miss rate in MARSSx86 is significantly higher than in the real machine, causing a significant performance mismatch.

To find the cause for this issue, we collected the number of competing blocks, defined as blocks that are of the same set, and their corresponding miss and hit numbers. Table V shows a snippet of the set breakdown. It is evident that competing blocks cause a significant miss rate per set. By profiling the miss address patterns, we find that the majority are from neighbouring addresses. Whenever a miss requesting block *N* occurs, then it is likely that the following misses are for blocks *N+1, N+2, N+3, N+4,* etc. The i7-920 tackles this problem by the support of its hardware prefetcher. Thereby, the capacity misses in the relatively small L2 cache can be avoided.

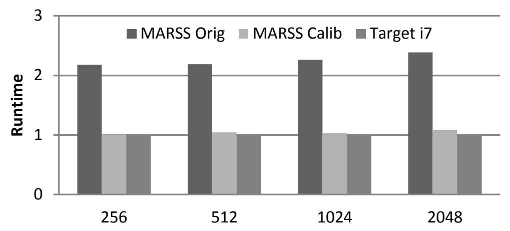


Figure 5. Normalized DGEMM runtime on original and calibrated simulators.

We subsequently implemented a stream prefetcher in MARSSx86 to model the hardware prefetcher in the real i7. Once a miss occurs, our model will aggressively prefetch 16 neighbouring cache blocks. As our results will show, using this stream prefetcher, we succeeded in improving the performance of the L2 cache to closely match the real system.

IV. EVALUATION

In this section, we discuss calibration results. We modified MARSS to account for the issues we covered in the previous section. Our calibrated MARSS is made available for download at [2]. We further evaluated the refined model to quantify the impact of this calibration on heterogeneous system design.

We use four applications that are representative in high-performance computing: GEMM, FMM [14], LU and Cholesky factorizations. We also evaluated the calibrated model on general-purpose SPEC2006 [13] and PARSEC [3] benchmarks in order to further validate its accuracy. In all cases, overhead of calibrations was observed to be negligible at less than 2% reduction in simulation speed.

A. Baseline Refinement

Figure 5 shows the runtime of GEMM for different input matrix sizes on the original MARSS, the calibrated MARSS, and the target i7. The original MARSS model predicts that the execution takes twice as long as in reality. By contrast, the calibrated MARSS model captures the behavior of the target i7 accurately with less than 5% error.

Figure 6 illustrates the accuracy improvements when running the five benchmarks with varying problem sizes for progressive calibration steps. *Orig* refers to the original MARSS, *Lat* to the latency-calibrated MARSS, *Lat+Pref* to the *Lat* version implemented with a simple stream prefetcher, and *Lat+Pref+Uop* to *Lat+Pref* with modified commit logic. A MARSS model with calibrated instruction latencies improves accuracy roughly by 10%. A stream prefetcher further boosts accuracy up to 40% overall. Finally, the fully calibrated MARSS model captures the behaviour of the target machine very accurately. It achieves near-identical runtime with the target i7, with less than 7% error across all HPC applications.

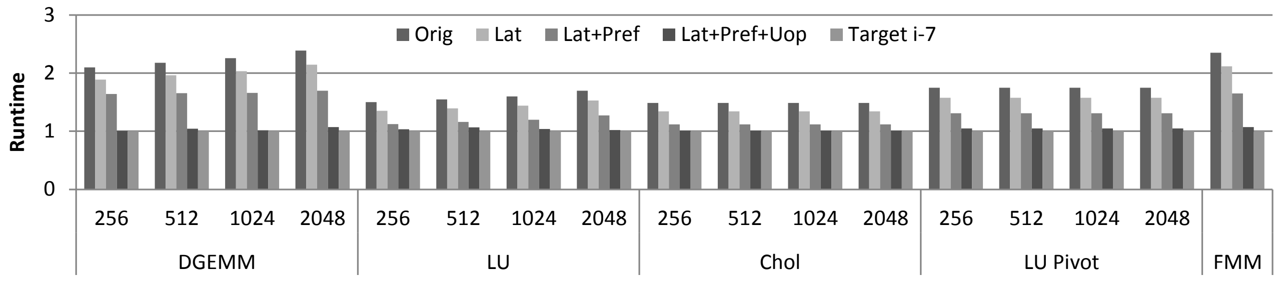


Figure 6. Runtime improvement of five benchmarks on each calibration step.

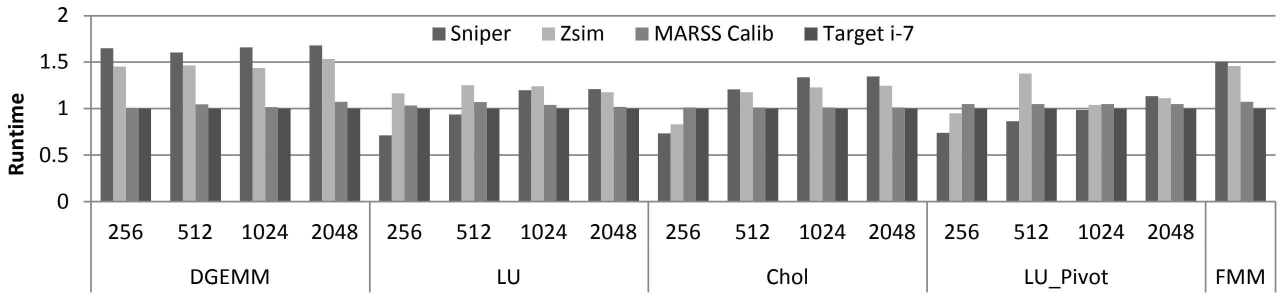


Figure 7. Normalized runtime of different simulators on five benchmarks.

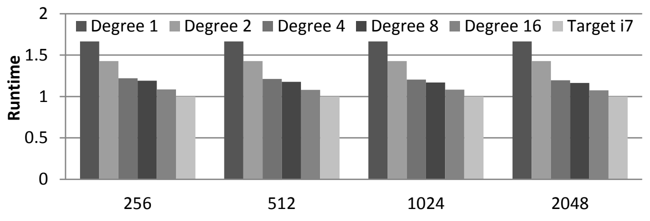


Figure 8. Normalized DGEMM runtime improvement as a function of prefetch degree.

We also observed the impact of the prefetch degree on the simulated performance. Figure 8 shows the normalized performance of the MARSS model employing different prefetch degrees. Increasing the prefetch degree generally increases the core performance and simulator accuracy. In particular, a prefetch degree of 16 on the MARSS model closely matches the performance of the targeted i7. This justifies our selection of realizing a stream prefetcher with a degree of 16.

Figure 7 shows the accuracy of our calibrated MARSS as compared to Zsim and Sniper. As mentioned before, gem5 is not able to run these benchmarks in their optimized forms due to its limited x86 instruction support. Among the supported simulators, the calibrated MARSS provides the best accuracy compared to the target machine. Zsim and Sniper in general are 20-50% less accurate due to their abstracted timing model. However, as a consequence, they possess a significantly faster simulator speed than MARSS.

To further evaluate the accuracy of our calibrated simulator, we compared performance when running SPEC and PARSEC benchmarks (Figures 9 and 10). We ran all C/C++ benchmarks for SPEC2006. For PARSEC, we excluded facesim, vips, and x264 benchmarks as they failed to successfully execute in MARSS. In this work, we targeted calibration for HPC applications. As results show, our calibration improves accuracy even for many general-purpose benchmarks while not significantly affecting results in cases where the original MARSS already matches well. An uncalibrated MARSS executes PARSEC and SPEC benchmarks with an average error rate of 19% and 9%, respectively. By contrast, the calibrated MARSS improves PARSEC and SPEC accuracy to 11% and 7% on average. Nevertheless, truly targeting other application domains would require applying our calibration methodology using representative microbenchmarks that would likely improve the accuracy even higher.

B. Impact on Heterogeneous Design

To quantify the impact of simulator calibration on heterogeneous system design, we integrated a publicly available cycle-accurate simulation model of a Linear Algebra Processor (LAP) [17] as an example accelerator into the system. We modified applications to offload the GEMM kernel operation to the LAP with the help of a device driver. The CPU thereby copies all the necessary data to the main memory before invoking the accelerator. The LAP then performs the GEMM and interrupts the host CPU when it is finished.

Figure 11 shows the overall system speedup when running GEMM, LU, FMM applications with all GEMM kernels offloaded to the LAP. An uncalibrated MARSS achieves average speedups of 10.3x when offloading a complete GEMM. However, when the calibrated core model is employed, the speedup is reduced to 5.3x on average.

For LU factorization, the application consists of three kernels, one of which is GEMM. When the problem size is relatively small, the speedup difference between the original and the calibrated model is small. However, as the problem size grows beyond 512, the time spent in GEMM rises significantly. In particular, for the 2048 problem size, the original MARSS estimates the speedup to be 4x. However, when the calibrated baseline model is integrated, the estimated speedup is reduced to 2.8x. If the accelerator was designed to have only a smaller gain by itself, due to the cost of integration overheads, an uncalibrated model could potentially project a system-wide speedup even though none or a negative one exists in reality.

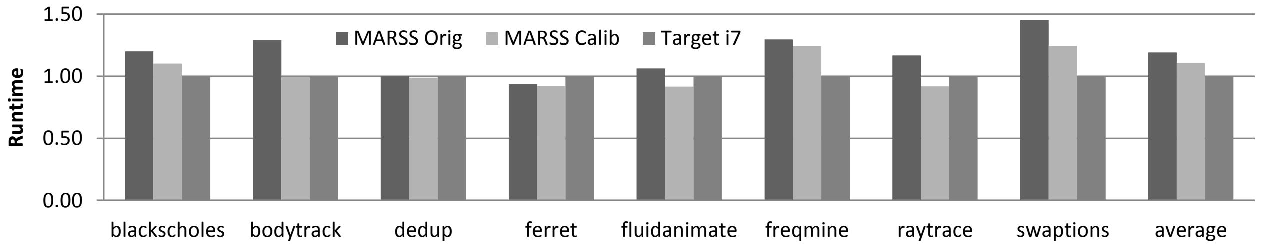


Figure 9. Normalized runtime of PARSEC benchmarks running on original and calibrated simulators, with respect to hardware platform.

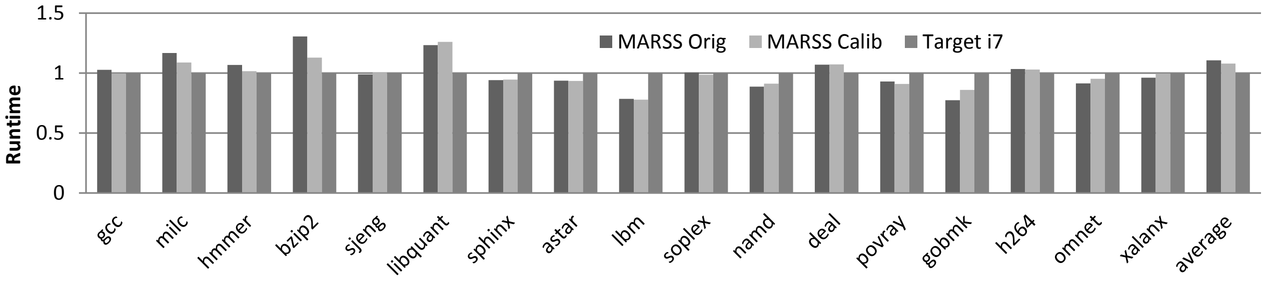


Figure 10. Normalized runtime of SPEC benchmarks running on original and calibrated simulators, with respect to hardware platform.

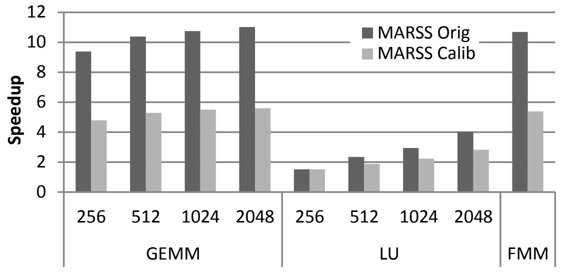


Figure 11. Speedup of benchmarks using original and calibrated simulators.

We observe similar behavior in FMM. Similar to LU, FMM consists of several kernels with one of them being GEMM. An unrefined MARSS core model reports a 10.3x speedup. However, using the calibrated model, estimated speedup is reduced to 5.3x.

V. Summary and Conclusions

In this paper, we presented a systematic methodology for calibration of cycle-accurate x86 simulators and the resulting impact on heterogeneous system design. We first described our step-by-step calibration methodology along with the major issues we found when applied to the MARSS full-system simulator. We calibrated the model to achieve a highly representative CPU baseline targeting HPC applications. Our calibrated MARSS is available for download at [2]. Results show it has on average less than 10% error including SPEC and PARSEC benchmarks. We further showed how an unrepresentative baseline CPU model can significantly over/under-estimate speedup when analyzing and evaluating heterogeneous system designs. This can result in misleading design decisions. In future work, we plan to build on top of this platform to perform trade-off analyses and heterogenenous system optimization.

Acknowledgments

This work was supported by NSF Grant CCF-1337393. We would like to thank George Biros, Robert van de Geijn, Hao Wu and Youssef Toubah for their help in preparing this paper.

References

[1] Intel software manual. https://software.intel.com/en-us/forums/intel-isa-extensions/topic/291765.

[2] MARSSx86-i7. https://github.com/LAProc/marss.

[3] C. Bienia et al. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[4] N. Binkert et al. The gem5 Simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, May 2011.

[5] D. Burger and T. Austin. The SimpleScalar tool set, version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, June 1997.

[6] A. Butko et al. Accuracy Evaluation of GEM5 Simulator System. In *ReCoSoc*, 2012.

[7] T. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation. In *SC*, 2011.

[8] C. Cascaval et al. A taxonomy of accelerator architectures and their programming models. *IBM Journal of Research and Development*, 54(5):5:1–5:10, September 2010.

[9] R. Desikan, D. Burger, and S. Keckler. Measuring Experimental Error in Microprocessor Simulation. In *ISCA*, 2001.

[10] H. Esmaeilzadeh et al. Neural Acceleration for General-Purpose Approximate Programs. In *MICRO*, 2012.

[11] A. Fog. Lists of Instruction Latencies, Throughputs and Micro-operation breakdowns for Intel. www.agner.org/optimize/instruction_tables.pdf.

[12] A. Gutierrez et al. Sources of Error in Full-System Simulation. In *ISPASS*, 2014.

[13] J. L. Henning. SPEC CPU2006 benchmark descriptions. 34(4):1–17, September 2006.

[14] D. Malthora and G. Biros. PVFMM:A Parallel Kernel Independent FMM for Particle and Volume Potentials. *Communications in Computational Physics*, 18(3):808–830, September 2015.

[15] P. J. Mucci et al. PAPI: A Portable Interface to Hardware Performance Counters. In *DoD HPCMP UG Conf.*, 2011.

[16] A. Patel et al. MARSSx86: A Full System Simulator for x86 CPUs. In *DAC*, 2011.

[17] A. Pedram, A. Gerstlauer, and R. van de Geijn. Codesign Tradeoffs for High-Performance, Low-Power Linear Algebra Architectures. *IEEE TC*, 61(12):1724–1736, December 2012.

[18] D. Sanchez and C. Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousands-Core Systems. In *ISCA*, 2013.

[19] F. van Zee and R. van de Geijn. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM TOMS*, 41(3):14:1–14:33, June 2015.

[20] Y. Yang et al. CPU-Assisted GPGPU on Fused CPU-GPU Architectures. In *HPCA*, 2012.

[21] Y. Zhu and V. J. Reddi. Webcore: Architectural Support for Mobile Web Browsing. In *ISCA*, 2014.