2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) | 978-1-6654-5954-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/ISPASS55109.2022.00032

Profiling an Architectural Simulator

Nedasadat Taheri, Alexander Manely, Ahmni R. Pang, and Mohammad Alian
University of Kansas

***Abstract*—In this work we set out to answer the following two questions: (1) where are the bottlenecks in a state-of-the-art architectural simulator? (2) How much can we make architectural simulations run faster by tuning simple system configuration? We choose gem5 as the representative architectural simulator, run several simulations with various configurations, perform a detailed Top-Down analysis of the gem5 source code, and tune system settings for running simulations more efficiently.**

I. Introduction

Software-based simulation is the backbone of computer architecture research and development. Since the inception of computer architecture as a field, many software-based architectural simulators has emerged. Currently, various architectural simulators are in-use by academia and industry for modeling different aspects of future computing platforms. gem5 [1], Sniper [2], MARSSx86 [3], and ZSim [4] are just a few examples of architectural simulators with currently active communities.

In this paper, we profile gem5 code and perform a detailed Top-Down [5] architectural analysis of gem5 execution to find the bottlenecks in the latest gem5 release. We use our profiling insights to perform simple system tuning to improve the performance of gem5 simulations.

This work is the first step towards better understanding the characteristics of gem5. Our major contributions in this paper are to answer the following questions: **Where are the bottlenecks in running gem5 on a Xeon server?** Our results show that gem5 is extremely front-end bound with large iCache and iTLB miss rates. Due to the huge code size, irregularity, and abundance of virtual functions and runtime polymorphism in the source code, there is no particular hot function or code block in gem5, the decoder unit in the out-of-order processor is under extreme pressure for supplying μOps for the back-end, and there are large miss prediction and resteer overhead in the front-end. **How much can we make gem5 run faster by adjusting system configuration and runtime tuning?** We show how backing gem5's address space using huge pages can improve simulation speed and energy efficiency. Our results show that enabling huge pages for gem5 simulations can improve the simulation speed by up to 27%.

II. Methodology

In this paper, we use gem5 as the representative architectural simulator and run simulation with various CPU type, number of CPU, and memory size. We use the following CPU types: *AtomicSimpleCPU (Atomic)*, *TimingSimpleCPU (Timing)*, *In-order CPU (Minor)*, *Out-of-order CPU (O3)*.

Table I shows the processor configuration when using Atomic, Minor, and O3 CPU models. We run full-system simulations with Linux kernel 5.4.0 and Linaro 7.5.0 toolchain.

TABLE I: **Simulation configuration.**

Parameters	Values
Core freq:	2GHz
Superscalar	3 ways
ROB/IQ/LQ/SQ entries	384/128/128/128
Int & FP physical registers	128 & 192
Branch predictor/BTB entries	BiMode/2048
Caches (size, assoc): I/D/L2	32KB,2/64KB,2/2MB,16ways
L1I/L1D/L2 latency,MSHRs	1/2/12 cycles, 2/6/16 MSHRs
DRAM/mem size	DDR4-3200-8x8/1, 2, 4GB
Operating system	Linux Linaro (kernel 5.4.0)

We refer to the experiments' configurations throughout the paper using a triple format as follows: *(CPU Type,Number of CPUs,Memory Size)*. For example, *(o3,2,4GB)* simulates a dual-core O3 CPU with 4GB of DRAM.

Workloads. We simulate the following workloads on gem5: `boot-exit`: Boot Linux on gem5 and immediately exit. `parsec`: We use several applications within the Parsec benchmark suite [6].

Physical Server Configuration. We run experiments on Dell Precision 7920 towers with Intel Xeon Gold 6242 CPUs, 20 physical cores (40 hardware threads), and 6 DIMMs of 16 GiB DDR4-3200 MHz (96GiB). We used the VTune profiler [7] for accessing the processor performance counters and performing the Top-Down analysis [5].

III. Profiling gem5

A. Function Diversity

As shown in Fig 1, there is no killer function inside the gem5 source code, that optimizing it can significantly improve the simulation time. As we increase the complexity of the CPU, the CDF of individual function execution time gets more flat; meaning that the hotness of individual functions gets lower. This is not surprising since as the complexity of simulation increases, more simulation objects get activated in each event to more accurately model the complexity of the hardware.

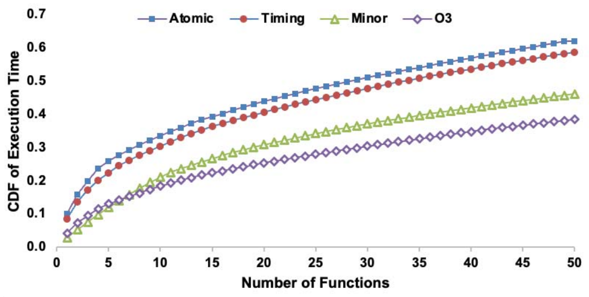

Fig. 1: **Top 50 hottest functions in gem5 simulating `parsec` with different CPU types.**

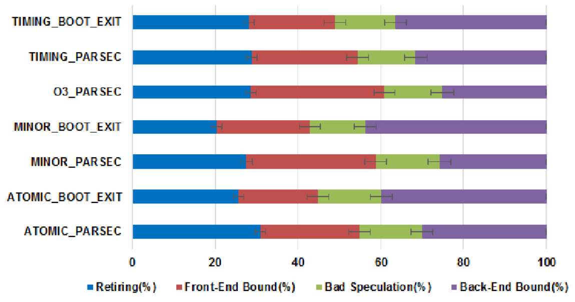

Fig. 2: **Top-level bottleneck breakdown of gem5.**

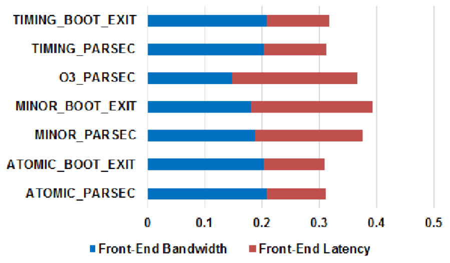

Fig. 3: **Front-end bound cycles breakdown of gem5.**

Therefore, more diverse functions get called when simulating with O3 CPU type compared with simpler CPU models. The total number of functions called throughout the simulation for the results shown in Fig. 1 are 1749, 3697, 4923, 6425 for Atomic, Timing, Minor, and O3 CPU types, respectively.

B. Microarchitectural Analysis

Figure 2 illustrates Top-Down analysis of gem5 simulating various workloads/configurations. Top-Down analysis splits the machine cycles into four categories: retiring, front-end bound, bad speculation, and back-end bound. We observed 20-30% of cycles retire instructions across different gem5 simulations. This is a relatively high retiring percentage compared to conventional workloads. For example, the average retiring cycle percentage for SPEC 2006 and Warehouse Scale (WSC) workloads is less than 20% and 15%, respectively [8]. However, the front-end bound and bad speculation is much higher relative to conventional workloads. On the other hand, the back-end bound cycles are lower than that of conventional workloads. The small dynamic working set size and temporally slow memory access to this working set results in having predictable data cache accesses from gem5 that can be efficiently captured by the hardware prefetchers or overlapped in the out-of-order engine of the modern processors. Also, the front-end bound cycles for gem5 are in the 20–33% range, even higher than that of WSC (large-scale services) workloads.

Figure 3 shows the classification of the front-end bound cycles between front-end bandwidth and latency. The main reasons for bandwidth and latency bound cycles are inefficiency in instruction decoding and iCache/iTLB misses, respectively. As shown in Fig. 3, simpler CPU models are more skewed toward bandwidth bound and as the level of CPU detail increases, the front-end becomes more latency bound. This can be explain by the fact that as the complexity of the CPU model increases, gem5 touches more simulation object binaries for processing each event. Therefore, the instruction cache footprint increases with the CPU model complexity and consequently gem5 becomes more front-end latency bound.

Along with iCache and iTLB overheads, we see a huge increase in the branching-related overhead when using O3 and Minor CPUs. The aggregated branching overhead for O3_PARSEC and Minor_PARSEC is 5.6× and 4.1× higher than that of ATOMIC_PARSEC. This is inline with the fact that increasing the CPU model's complexity results in more function calls, parameter checks, and event generation and activation. These in turn increase the branch density of the code and contribute to the large branch overhead and increase of hard to predict branches.

IV. Improving Simulation Efficiency

As discussed in detail in Sec. III-B, due to the large instruction footprint of gem5, we observe a lot of cycles stalled on iTLB misses while running gem5 simulations. A simple solution to the iTLB misses is to use huge pages to back gem5 code/text. Linux supports Transparent Huge Pages (THP) [9] that is a feature that provides transparent huge page allocation for user applications. This means that THP does not require applications to be modified or even be linked to a library at runtime. However, the current THP implementation only works with anonymous memory mappings (i.e., the memory that is not backed by the file system such as implicit memory allocation in the heap and stack) and tmpfs/shmem. Therefore, THP does not back the code (text memory segment) of an application binary using huge pages. We use libhugetlbfs [10] that allows applications to back text, data, dynamically allocated memory, and shared memory with 2MB or 1GB huge pages. Unlike THP, libhugetlbfs requires linking applications at runtime to the libhugetlbfs library. The libhugetlbfs library will intercept the memory allocation calls from the application and use huge pages if the allocation size matches the huge page granularity.

Figure 4 shows the simulation time when using huge pages, normalized to baseline gem5. As shown, this simple optimization, without any source code modifications, improves the simulation time by up to 27% across the board.

V. Conclusion

In this work, we performed a detailed Top-Down microarchitectural analysis on gem5. Our analysis reveals three main bottlenecks in gem5 execution: (1) high iCache and iTLB misses, (2) high branch resteer overheads, and (3) extremely

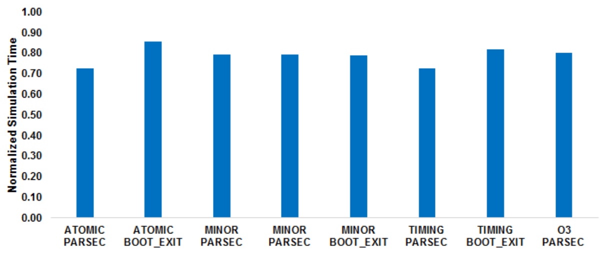

Fig. 4: Normalized simulation time when backing gem5 code with huge pages.

low μOp cache utilization. These bottlenecks are the result of huge code size, cold code execution, and extensive use of virtual functions, and polymorphism throughout the gem5 source code. Simply backing up gem5 source code using huge pages provides up to 27% reduction in simulation time.

REFERENCES

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, 2011.

[2] W. Heirman, T. Carlson, and L. Eeckhout, "Sniper: Scalable and accurate parallel multi-core simulation," in *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES-2012)*. High-Performance and Embedded Architecture and Compilation Network of ..., 2012.

[3] A. Patel, F. Afram, and K. Ghose, "Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors," in *1st International Qemu Users' Forum*, 2011.

[4] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," *ACM SIGARCH Computer architecture news*, vol. 41, 2013.

[5] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[6] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors," in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[7] "Intel® vtune™ profiler," https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html#gs.jescps, 2021.

[8] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2749469.2750392

[9] A. Arcangeli, "Transparent hugepage support," in *KVM forum*, vol. 9, 2010.

[10] "libhugetlbfs," https://github.com/libhugetlbfs/libhugetlbfs, Accessed Dec. 2021.