

Cache in RISC-V single CPU simulator: QEMU vs gem5

Julie Dumas

Univ. Grenoble Alpes, CNRS,
Grenoble INP, TIMA
38000 Grenoble, France
julie.dumas@univ-grenoble-alpes.fr

Joachim Le Fournis

ENSIMAG
Grenoble INP, TIMA
38000 Grenoble, France
joachim@le-fournis.fr

Abstract

QEMU is very fast to simulate single core and even more for multi-core/manycore systems. It is able to add on top of its simulation system a cache simulation by probing instructions executed and memory accesses. However, due to the fact that it simply probes accesses, it is fast from being perfectly accurate, and we found out a lot of differences in all simulation type. We managed to explain almost all of them in bare metal, but it is only the top of the iceberg: there is quite a lot more to dig in linux Full System and Syscall Emulation simulation. We already explored some ideas (some of which didn't brought us interesting results), but there is still a lot more to do in these simulations. For a software engineer comparing algorithms, QEMU should be perfectly adapted in all modes of simulation (easy to use, performant, even if it is not perfectly deterministic), but the architecture researcher should be really aware of the important differences it has compared to gem5, mostly in Full System and Syscall Emulation.

1. Introduction

Caches have played an important role since their introduction in improving execution time by reducing time spent on instruction fetching and data access. They act as an intermediate memory between the CPU which tries to load/store from/into a given address and the memory below it (which can be a bus connected to DRAM for instance, or another level of cache). By allowing faster access (about half a ns in L1 caches, the smallest and fastest ones, compared to hundreds of ns for the main memory), it speeds up the execution of a CPU, at the trade off their capacity.

All of this complexity and efficiency lead software engineers to adapt their code in order to use most of the performance improvement caches offer (such as in matrix algorithms [1]), and thus they wanted to get statistics, notably for data caches hits/misses/missrate but also the addresses of instruction causing the misses. While these datas can partially be accessed when their software is running on a real computer through performance counters, such as the ones introduced in RISC-V [2], to get all of them they should have acquired expensive hardware to probe the cache directly, leading them to prefer cache simulator, which could be easily installed freely and get them all the statistics they wanted.

However, software engineers were not the only ones interested in simulating cache: researcher specialized in processor architectures could be willing to change the cache architecture simulated (ie the number of core simulated, but also the size, associativity, replacement policies, ... of the caches, and the way they're connected, with different number of layers and cache coherence for instance) to study the performance variations [3], or also to test new ideas of cache coherence protocols that don't exist yet on real hardware for instance.

We chose to study systems based on the RISC-V ISA, because it has gained a lot of popularity within the system architecture academia, thanks to its young age allowing it to be in continuous development, but also its royalty free design. One of our main target being archi-

tecture researcher who would use a lot of cache simulations, we will stick to this architecture in this article.

The goals in this paper will be to see in which conditions can QEMU be used in replacement of gem5, which will be our reference for the results (as it is in architecture academia), and we will try the most we can to explain the origins of the differences.

This paper is organized as follows. We first outline in Section 2 different possibilities of simulating cache, with their pros and cons, after a short reminder on cache mapping. In Section 3 the different environments of our simulation (the benchmarks and way to run the benchmarks on the simulators) are described. Section 4 shows and explain the results of the simulation we ran on unmodified QEMU and gem5. We describe partially in Section 5 the differences we were able to pinpoint in both simulator source code, while giving other results found which don't explain the outcomes, and give ideas of the possible sources of differences we weren't able to identify. Finally we conclude in Section 6 while giving ideas of other experiment which could be interesting.

2. State of the art

2.1. Cache background

There are multiple ways to store instructions and data in cache which are directly related to physical problematics [3]. Firstly, instead of storing the machine word size into cache (here on RISC-V it would be 64bits), the cache will store an aligned chunk of the main memory of a specific size named cache line size (shortened BPL, Bytes Per Line, which is 64 bytes in this paper). Secondly, the cache can't store all the memory below it (top of the line consumer hardware can have at most 1MB L1 cache, whereas it is getting common to have 32GB of memory). This introduces different strategies, so called mapping (because they link a memory address to one or more specific places within the cache), among which we will focus here on the two most common ones:

- N-way associative caches are organised in sets of N cache lines, where multiple can be stored in. To decide where an address will go, we split it into three parts: an offset, which indicates where within the cache line this byte is, an index, which tells which set this address belongs to, and last but not least a tag, which will be stored next to each cache line content, allowing the cache to know at every time which address every bytes stored in it correspond to. On Figure 1 for instance, we have a 8bit system with 4 BPL and 4 sets. You can see that every pack of 4 bytes of the main memory on the right can be stored only in one set, and on the bottom how a specific address (0x96) is split in order to know in which set it will be stored. Another important part of the N-way associative cache is that when we have a miss (a data not present in the cache), and if all cache lines of the set linked to this address are used, it will have to chose which lines to remove from the cache. This is called an Replacement Policy (RP), and we will use one of the most common one in this article: Least Recently Used (the cache line that hasn't been accessed in read/write for the longest time will be evicted). A cache of sizeMB and BPL Bytes Per Line, with N ways and s sets will verify

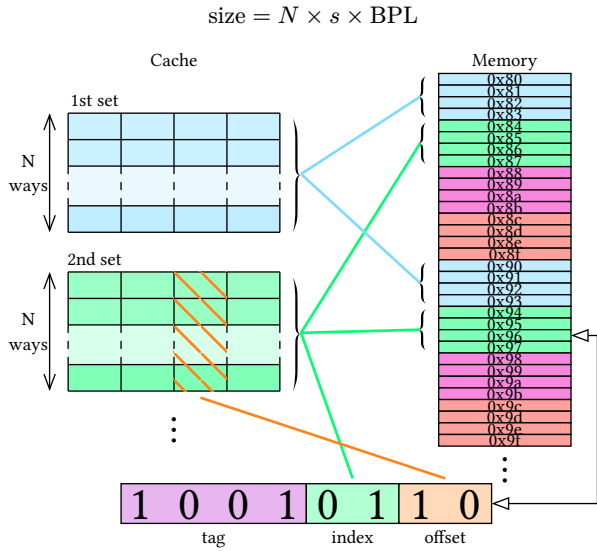


Figure 1: N way associative cache with 4 BPL, 4 sets: every address in memory is split in 3 pieces, which corresponds to their place within a cache line (offset), the set they are in (the index) and informations that will be stored in the cache to find which address the line corresponds to (tag)

- Direct mapped cache is a 1-way associative cache: every address can only be found at one place of the cache, which corresponds to its index. It is the simplest to construct, but is the least efficient one [3]. This cache mapping moreover doesn't require any RP, because when an instruction / a data is demanded by the CPU, if it isn't already in place at its only possible position, the cache will get it from the memory below it, and remove the data already at its place to store it in the cache.

2.2. Cache simulators

As other surveys [4] have shown, there exist a large variety of cache simulators with different properties, such as the type, the level and the scope of the simulation, on which we will focus here.

While it is technically possible to simulate a real RISC-V CPU with its cache at a transistor level in order to get the most exact statistics as possible, it would be painfully slow for bare metal programs, and a living nightmare to run a linux in this simulation. That's why we won't go deeper in these types of simulation.

Some of the simulators listed, like Cachegrind, run as an application on an os, and simply probe the access to the memory of the program given as an argument while running it on the OS directly. While being easy to use in a daily life of a software engineer, it clearly isn't adapted to what we're doing here: the RISC-V CPUs running linux smoothly aren't that common, even in architecture laboratories. That's why we need to simulate the whole system, which corresponds to more than a cache simulator (M) scope in [4]. Moreover, as we will explain in Section 3, we don't want to simulate only linux executable using the host libraries to answer syscalls of the program, but also a whole system, with or without its own OS. That's why our simulators should have level supported in [4] Application and Full System. Last but not least, we would like to have open source simulators which are still updated and used by other researchers.

This drastically reduced our possibilities to the two simulators here shortly described:

- gem5 [5], [6] is a famous hardware architecture simulator, which is categorized as timing typed: this means that it doesn't simply try to produce what the binary run on a real machine would produce, but moreover will track the time taken by each steps of its internals (for instance the caches in its code must specify how

much time each hit takes), which allows it to reproduce faithfully a whole system. It is perfectly deterministic, each simulation in a same environment will produce the same results in statistics, without having to simulate each transistor. However, this simulator is still not that fast: when using only 1vcpu it is about 10 times slower than QEMU, but it is worse when running multicore/manycore programs because it can only use one core of the host.

- QEMU [7] is at the opposite side of the type spectrum (even if not mentioned in [4], its features of cache simulation have been introduced since), because it is considered as a functional simulator: its main target is to get the same outputs from the software run, and not to get the same timing. That is why at first glance it shouldn't be a good solution to get statistics about cache, which is intricately linked with the way the hardware works. However, due to its speed up compared to gem5, which is tenfold in multi/manycore applications, because it can use multiple CPUs of the host machine, it is a solution which is interesting with its feature to allow researcher to add plugins to the Tiny Code Generator (the Dynamic Binary Translator of QEMU, which converts target instructions just in time to host instructions), such as simple cache, cache coherency, more efficient L1I [8], ... simulations.

3. Simulation environments

3.1. Benchmarks

For this comparison, we need programs to run on the simulators, which optimally should try to test a lot of different features of the caches. That's why we need benchmarks which text section and data/rodata/bss section should be bigger than the cache, so that eviction in both simulators is also tested. That's why our choice stopped on two benchmarks:

- PARSEC [9], [10] is designed to study on chip multiprocessors with shared memory, which was the starting point of this study, and which we will discuss shortly in Section 6. Moreover, it is a reference benchmark in the industry, which influenced us to keep it. It is suited to test a large spectrum of features from shared memory systems, among which cache performances. We chose to stick with 4 of its benchmarks: cholesky, fft, lu_cb and radix (each one is $\geq 45\text{kB}$ in text and $\geq 50\text{kB}$ in data/bss/rodata).
- embench-iot [11] is a benchmark based on real life application (qrduino, pjpeg, ... and a lot of crypto libraries like crc32, aes, sha256, md5, ...) made specifically for embedded system, which is partially positive for us: it doesn't require standard library except for classical memory function (memset, memmove, strlen, ...), libgcc and libm, which ensures us that it won't use black box (such as memory allocation) which could be source of randomness between simulations (allowing us to expect exactly the same results between QEMU and gem5 but moreover between QEMU simulations). However, in order to have a sufficiently large binary ($\approx 180\text{kB}$ in text and ≈ 60 in bss/data/rodata), we needed to put all the programs in one single binary, which we will run multiple time during one simulation. This allows us to test a large variety of behaviour of the cache but it still provides us a binary which has a strong locality (the programs run don't depend on each other, so instruction misses only occur when switching programs, and most of the data accesses corresponding to either the cache, rodata arrays or dynamically allocated memory, which are contiguous with the basic allocator implemented).

3.2. Simulation types

Both of the simulator support two simulation types, which motivated our choice for them:

- Full System (FS): the simulator simply fetch, decode, execute the instruction as they are stored, and has to implement features of usual RISC-V hardware (such as configuring the MMU, add

timers and interrupt controllers, UART, ...). This simulation mode will be declined in two different ways:

- Bare metal: we will launch a benchmark directly by executing its c code after a few steps to configurate everything needed (in our rt0, which doesn't require openSBI, we enable Floating Point Unit, set stack/global pointers, disable any Address Translation and Protection and clear bss). Embench-iot only run on top of this setup.
- Linux with busybox: we built linux for RISC-V and have disks with the benchmarks launched after the kernel booted. This is supposed to lower variability of the simulation, because in theory there should be no influence between host and target, but is far from being easy to use (you need one disk image with the benchmark and its inputs, and busybox for each benchmark, and we need to copy them before executing the benchmarks in parallel).
- Syscall Emulation (SE or User Mode in QEMU): the simulator fetches an instruction from the memory, decodes it and when it meets a syscall instruction (ecall on RISC-V) it uses a specific table for the simulated ISA in order to execute the host code of the simulator which reproduces the behaviour of a linux kernel running on RISC-V. This has a lot of pros, such as its ease of use: when you have cross compiled your program to a linux RISC-V environment, you can launch it directly, it will have access to all the files on your computer without any troubles, and drastically improves performances compared to the linux FS, but it also has drawbacks, because it greatly depends on the way the simulator implements these syscalls for instance, which has no reason to be standardized between simulators. We also ran PARSEC benchmarks on top of it.

3.3. System simulated

	QEMU	gem5
vCPU	1 vCPU	1 TimingCPU
L1I/L1D	16KB, Least Recently Used RP, N -way associative (N will change between executions)	
L2	Not detailed here, 64kB LRU, direct mapped	
Peripherals	Syscon, UART, Platform Level Interrupt Controller, Core Level Interrupt Controller	
machine, platform	virt	HiFive

Table 1: Short description of the simulation environment

During our first simulation we found out that cache associativity was heavily influencing the differences, which lead us to decide that our cache associativies will change during the experiments. However, the cache sizes, RP and the number of vCPU will stay constant in this article. On FS, we chose to stick with a conventionnal RISC-V machine, which is already implemented in QEMU as the virt machine (with RISC-V interrupt controllers, UART for the console and Syscon to exit looping simulation). In gem5 most of this was implemented in the HiFive platform, except for the Syscon, which we implemented so that we can have the same binaries between QEMU and gem5.

Because some of the results we will see next depended on the host runtime, here is the configuration of the host machines:

Simulation	Configuration
embench	Precision 3591 (Intel(R) Core(TM) Ultra 5 135H, 16GiB of memory), Ubuntu 24.04
PARSEC (FS/SE)	1 x AMD EPYC 7F72 24-Core/48-Threads (Zen 2) @ 3.2Ghz, 192GiB of memory, Debian 11

Table 2: Host machines

4. Results

In this section we have a lot of graphics such as Figure 2: we ran the same programs on gem5 and 20 times on QEMU in order to compare them (because unlike gem5, QEMU isn't usually deterministic), and extracted for each simulator a count of access and of hits of each cache. We computed the geometric mean [12] of the QEMU simulation statistics and compared it to the gem5 result using the SMAPE [13] formula:

$$\text{SMAPE} = \frac{100}{n} \sum_{i=1}^n \frac{|F_t - A_t|}{(|F_t| + |A_t|)/2}$$

where F_t and A_t respectively correspond to QEMU and gem5 values of different benchmarks. This allows us to get a single statistic describing the difference on a scale of $[0, 200\%]$. The color bars on the side of each subplots show this percentage, and you should pay attention to their scale, which will change between each subplot and plot.

We made this while changing the associativity of L1I and L1D in $\{1, 2, 4, 8, 16\}$ between each simulation of the same benchmark, giving us the image seen in Figure 2.

Last but not least, we also traced in Figure 6 for instance the coefficient of variation of the same graphs, which is $\text{CV} = \frac{\sigma}{\mu}$ (where σ is the standard variation and μ the arithmetical mean).

4.1. Bare metal

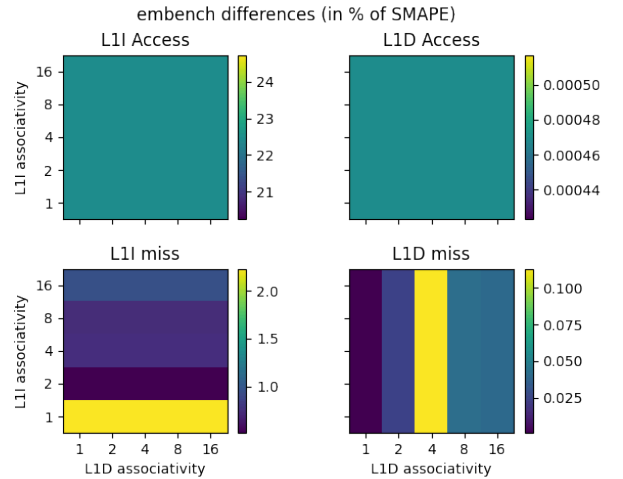


Figure 2: Differences of embench on bare metal

Firstly, we can see on Figure 2 that there is a difference in the number of access of both caches, which is way more important in L1I (we will explain that in Section 5.1), but this difference doesn't depend on the associativity of the caches (that is coherent: the cache associativity shouldn't influence the CPU, which is the only one which can influence back the number of access).

Also, another important part is that the associativity of a cache doesn't influence the hits of the other one (clearly seen with the horizontal stripes on the L1I corresponding to a same instruction associativity but different data associativity, and the same happens

for L1D). For the same reason as before, this is what we would expect of all simulations.

However, there isn't a clear correlation between hits differences and associativity: on L1I it is maximized when we have a direct mapped cache, but on L1D it is minimized when direct mapped and maximized when using an associativity of 4. It is nonetheless the type of simulation with the most predictable results.

4.2. Linux Full System

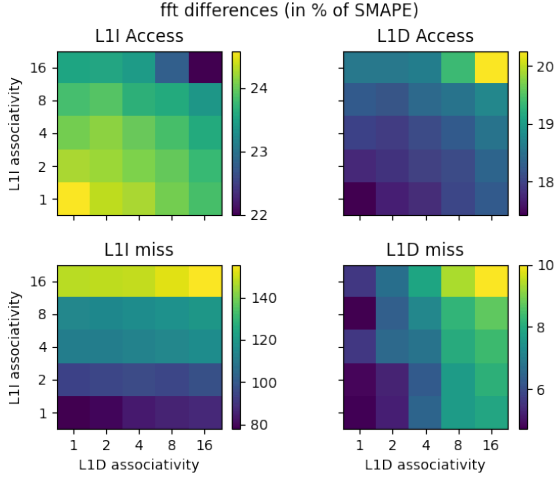


Figure 3: PARSEC fft FS differences

On all benchmarks we found similar graphics to Figure 3, where the differences are maximized / minimized on the corner (ie the closest you get to a fully associative cache or a direct mapped one). As you can see, there is quite a lot more differences in the L1I hits and L1D stats compared to the bare metal results.

One benchmark stands apart: Figure 4 which contrary to Figure 3 (and the other one) hasn't an opposition of more differences in L1I access means less differences in L1D access, and the opposite for the difference in hits differences in L1I/L1D. We weren't able to explain that, and it somehow warned us about the visible patterns that looked emerging of these graphs which aren't always universal.

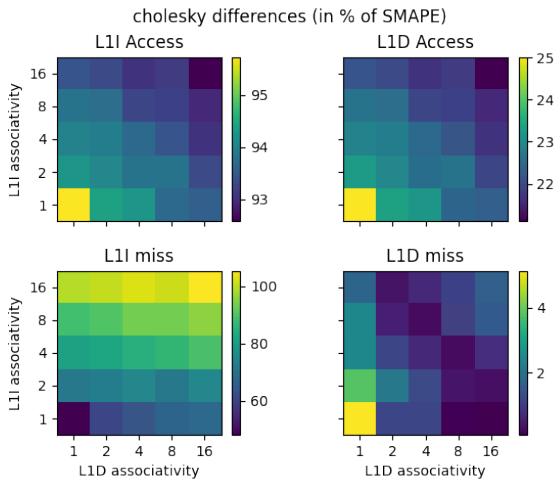


Figure 4: PARSEC cholesky FS differences

Also, we can also notice that the associativity of one cache clearly influence the number of access of the other one, without massive influence however on the differences in hits. This could seem contradictory with what we said earlier, but we plotted in Figure 5 the number of access/misses on gem5 running cholesky benchmark in FS, and as you can see, here there are differences too. We didn't search much to find explanation about that, but the most evident answer is that the associativity of a cache would cause different

misses, which would cause different timings in the gem5 simulation, causing the linux kernel to behave differently. QEMU not being able to simulate the timings caused by cache misses would necessarily, as expected, cause difference with gem5.

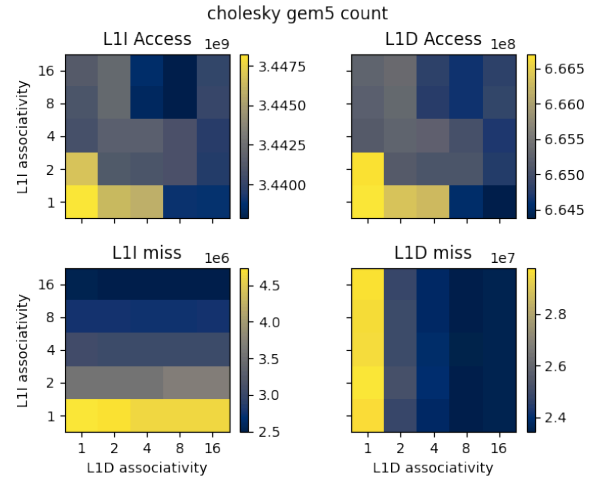


Figure 5: PARSEC cholesky gem5 FS counts

Also, we can notice in Figure 6 that even if there is quite a lot of variation between QEMU simulations compared to the coefficient of variation in SE, the cache associativity doesn't seem to change that much the coefficient of variation (at most it doubles for the same statistic), and there isn't a specific place where the variation would be minimized or maximized.

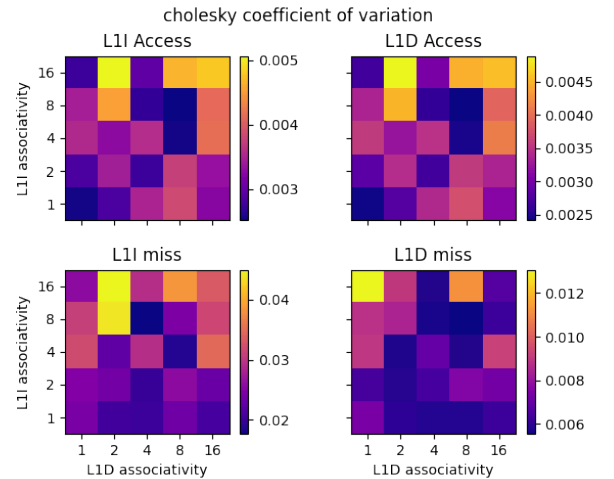


Figure 6: PARSEC cholesky FS variations

4.3. Syscall Emulation

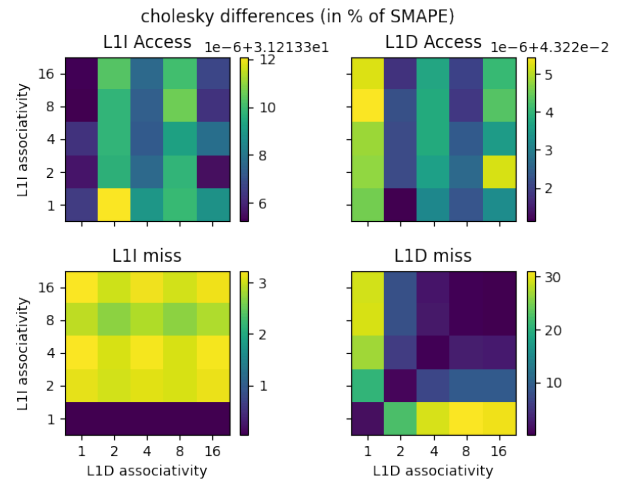


Figure 7: PARSEC cholesky SE differences

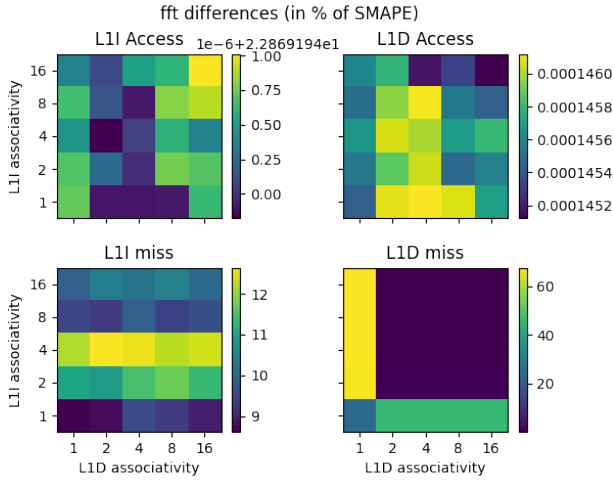


Figure 8: PARSEC fft SE differences

In SE, the results graphics on demands seems way more chaotic, but it could be explained because the differences are still important (on our benchmark we have from 23% up to 54% on L1I, and from $10^{-4}\%$ to $10^{-2}\%$ on L1D), but within the same benchmark, the differences on number of access are almost always the same with minimal variations (which explain the somehow random shapes appearing on the top of the graphics). Meanwhile, the differences in hits are also important, but the variation is even more important (with a pattern on the L1D hits differences, which are maximized on direct mapped instruction or data cache). This lead us to think that there may be a difference with the address the program (or its libraries) is loaded to.

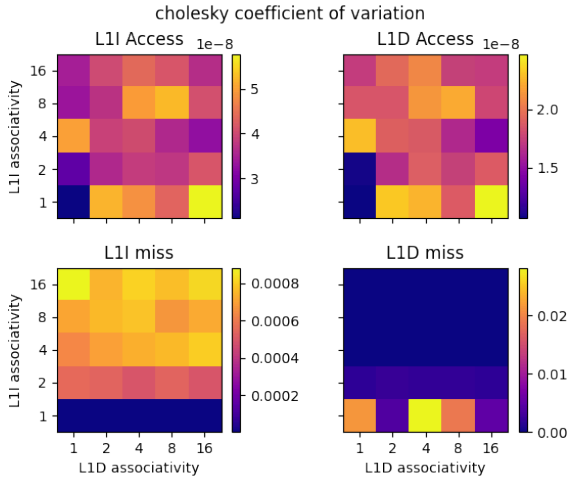


Figure 9: PARSEC cholesky SE variations

We can also notice in Figure 9 that QEMU is way more stable on all statistics (ie gives the same results between executions) compared to the FS ones, except for the number of L1D accesses which is very slightly more instable.

5. Origin of the differences

After having run the benchmarks without any changes on QEMU and gem5, we tried pinpoint differences that explains these previous results, with different ways: we tried to add debug logs on cache and CPU operations to understand how it worked, and we tried to diff the gem5 with the QEMU ones on bare metal simulations, that we chose because of the small variations on QEMU and the short time of the execution. However due to the large files, we weren't able to find all the origins of the difference even in bare metal. We will nonetheless describe in this section the ones that we managed to pinpoint and simulation differences that didn't cause the previous results.

5.1. Instruction alignment

The most important difference we managed to locate was an issue occurring with compressed RISC-V instruction. While some other RISC architectures, like ARM [14] or MIPS clearly specify that the instructions must be aligned on their size, RISC-V, on the opposite, can mix compressed and uncompressed instructions without the need of a specific instruction between them, which allows situation where an uncompressed is not 4 bytes aligned.

This is not in principle an issue, because this works great on real hardware and avoids issues raised by ARM thumb or microMIPS of frequent switches from one mode to the other by using an additional jump. However, the issue comes from gem5 which for historical questions (related to decoding variable sized x86 instructions), expects RISC-V uncompressed instructions to be aligned (like the others RISC ISAs simulated), or it will produce two accesses to the cache: they will get the word-aligned words which the instruction respectively begins and ends (for instance, an 4 bytes instruction at address 0x02 would produce a demand on the cache at address 0x00 and 0x04).

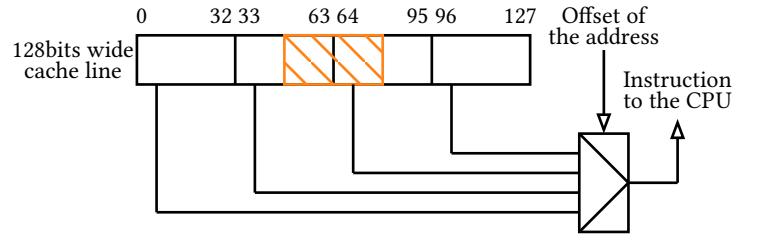


Figure 10: Hardware simulated by gem5: to get the 4 orange bytes at the offset 6 bytes, the CPU would have to start 2 transactions (at offset 4 and 8 bytes) with the instruction cache

While hardware could implement instruction fetching like this (as in Figure 10), it seems pretty unlikely to be done on real hardware due to the frequency this event happens. On Figure 11 we changed QEMU to produce the same accesses during unaligned 4 bytes instructions on bare metal simulations (even if QEMU seems to implement it correctly, we chose to change it to look like gem5, because it was simpler than the opposite).

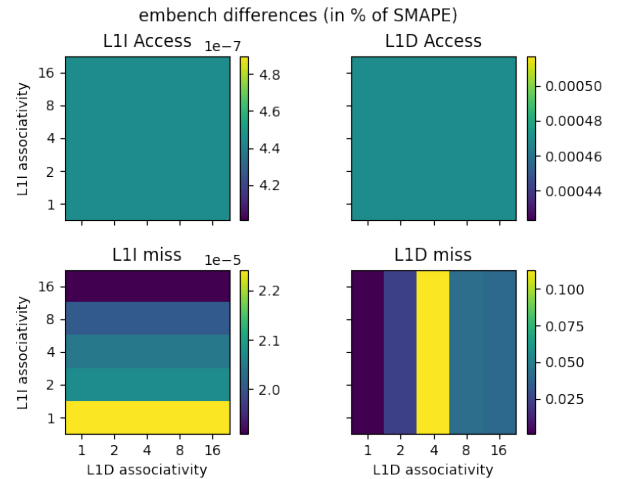


Figure 11: QEMU cache plugins changed in bare metal to produce the same behaviour as gem5 on 4-bytes unaligned instructions

5.2. Simulation loop exit

While Section 4.1 may have depicted QEMU on bare metal as perfectly deterministic, we found something that was a little bit awkward: on some simulation, we would get a few accesses in differences between QEMU simulation. This can be seen on Figure 12.



Figure 12: Two values that differ from the other on bare metal

When experimenting to understand in which circumstances this would happen, we found out that it depended on the workload of the host, giving us different results with different probability depending on the number of simulation processed. We summarized this for one experiment in Table 3 where for each possible outcome ($\lambda + 0$, $\lambda + 5$ and $\lambda + 10$ L1I hits, where $\lambda = 1254240576$), we give the percentage of apparition of this outcome depending on the number of simulation launched.

$\lambda = 1254240576$	< 20 sim	= 50 sim	= 100 sim
$\lambda + 0$	100%	82%	48%
$\lambda + 5$	0%	18%	46%
$\lambda + 10$	0%	0%	6%

Table 3: Probability of seeing $\lambda + k$ L1I demands depending of the number of simulations

We found out that the event to exit the simulation loop is dependant on the host OS: when the specific byte is written as syscon address to power off the CPU (the solution we took from QEMU implementing sifive hardware and which we added to gem5), instead of directly exiting the simulation loop like gem5, the few next instruction (5 or 10 on the previous exemple) can sometimes be executed (because we forgot to add a `for(;;) { asm("wfi"); }` to limitate this effect).

5.3. Split across cache lines

Another issue related to Section 5.1 is the situation where the instruction fetched or the data accessed address and size requires to read/write from/to multiple cache lines (for instance when reading 4 bytes at address 62 in caches that have 64 bytes cache line size). QEMU is implemented in the most basic way, by not taking into account the size of the accesses, while gem5 works well, as long as the transaction need to access at maximum on two different cache lines.

Cache line 1:	Data 1	Data 2	Data 3
Cache line 2:	Data 3	Data 4	Data 5

Table 4: Example of a data split across multiple cache lines: accessing data 3 would require to access to the cache

This issue doesn't happen on instruction caches on RISC-V due to the changes made in Section 5.1 (which fixed this issue on L1I), but we tried changing the L1D behaviour to mimic gem5, which gave us Figure 13. However, this doesn't bring real changes in the results: even if RISC-V allows data accesses to be misaligned (ie not aligned on the size of the access), but the aligned ones are usually faster, so most of the compilers will try to only generate only aligned ones, so that this situation will never happen.

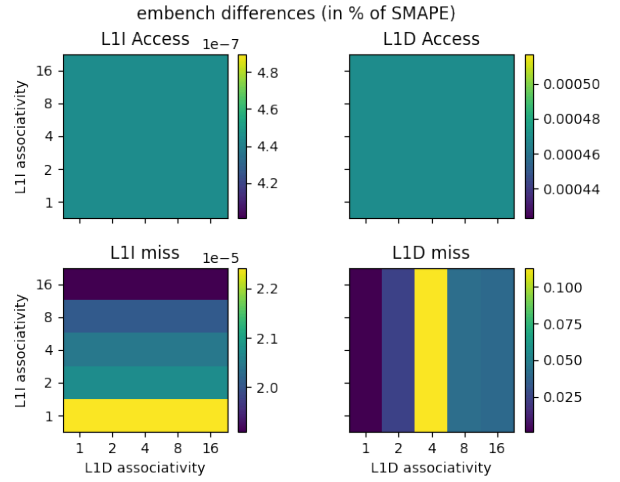


Figure 13: Cache statistics after fixing Section 5.1 and data split across lines

We found out that on our embench-iot bare metal simulation, we didn't have any differences past this point on the number of accesses on both caches, except for the boot up (where QEMU seems to access a few bytes at a specific address), and at loop simulation exit (which can cause just 1 wfi more in QEMU than in gem5).

5.4. Host run-time influence

An issue that has already been mentioned in FS simulation is the influence of the host runtime on the simulated one: when the target sets up timers, as in the kernel simulated (within the `update_cfs_group` function), the alarm will be triggered based on the host RTC (using on x86 `rdtsc` for instance). That can generate differences in the task scheduler of the kernel, adding randomness to the statistics of the simulated cache. This has been more documented in [8] and [15], which we advise you to read if you want to learn more about it.

5.5. Other seen differences

While trying to pinpoint the origins of these dissimilarities in SE, we found differences that we didn't have time to ensure how they influenced the simulation or that didn't change the results:

- Peripherals aren't accessed the same way in gem5 and QEMU : while in gem5 it is standard that the cache is placed inbetween, it looks like QEMU isn't simulating cache when the address corresponds to peripherals.
- There is different stack pointer alignment within a cache line at software boot in SE, because the top of the stack region wasn't at the same alignment and because the executions informations (among which envp and argv arrays, and sizes and pointers to these arrays) were placed above the stack pointer differently before starting it. We were able to confirm that this didn't have a massive influence on the results, and that other factors in SE are very likely more important.
- We found during our experiment on bare metal that QEMU was translating addresses that shouldn't be translated: we disabled the Memory Management Unit (it is set to bare), but we still found different addresses between QEMU and gem5, that disappeared when removing in QEMU every mention of `hwaddr` to only use `vaddr`. We didn't have time to study more deeply the reason which lead QEMU to wrongly translate the address, but it could be the origin of differences in misses. We tried on QEMU cache plugin to change every mention of `hwaddr` to `vaddr` on bare metal programs, but it didn't seem to change deeply the statistics, maybe the translated addresses are still aligned the same way. There is more to study here, but we didn't have time to.

- Last but not least, it seems that the gem5 misses counter is sometimes wrongly incremented: we tracing every increment of it, we found some of them done due to write accesses of 64 bytes causing a miss, which correspond to the BPL size. This lead us to think that the cache misses probably count twice: one will happen when the CPU produce an access to the cache, the other one when the memory below answers the cache to write the cache line in it.

6. Conclusion

The results we got above are not perfectly clear with our targeted usage (of software engineer and architecture researcher), because we found out that while bare metal seems the most correct simulation type to have accurate mesures (with perfect number of access for instance), there are still things to dig (Section 5.5), and it is not the easiest way to run a program. The Full System simulations have important differences compared to other types of simulation, and will produce important variation depending on the cache associativity, but these variation are somehow predictable (they increase of the diagonal), and is a little bit easier to use. The Syscall Emulation (or User Mode) simulation have way less differences that FS, and smaller variations too, while being easier to use, but these last ones are way less predictable because of the implementation of the syscalls within the simulators, which have no reason to be really predictable, nor to be the same between QEMU and gem5. In global, except for the bare metal simulation (if the issue of translated addresses is fixed), QEMU should totally be usefull even for very precise works, like the ones of architecture researcher. However, for the other simulation types, they should stick to be used only when comparing executed programs and not slightly different hardwares.

Another important point to highlight of this work is that there isn't always a universaly good implementation of cache simulation (like in Section 5.1), and that even if gem5 is more accurate (thanks to timing typed [4]), it is not always the ideal simulator.

There are a few points described in Section 5.5 which would be the last important points to study about this environment, but there are more systems that the one we studied here that QEMU would improve to simulate, among which different caches properties (prefetching, replacement policies, clusivity, write back policies, ... or even different cache plugins such as MOESI ones used here at SLS team in TIMA laboratory) the ones with a lot of vCPU (multicore - manycore). This work has already been started by Jules Bodin (intern in 2024 at SLS), but there were a lot of explanation that were still missing, and these are the upcomming most important ones.

To conclude, it has been difficult to really understand all the differences between gem5 and QEMU because they are numerous, discrete and on 1vCPU bare metal programs (which should be the easiest differences to track, because it is deterministic and small), it is very hard to understand (when trying to generate traces on QEMU and gem5 on small execution, we got multiple GBs of files which took tens of GBs of memory when trying to diff them). Some other ways to understand differences between QEMU and gem5 would have been possible (if we stick to bare metal to find differences, trying to connect to both of the simulators with their gdb interface and step execute them until we got a difference would have been a possibility), but this is uncertain this would have produced more interesting results.

Bibliography

- [1] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, in ASPLOS IV. Santa Clara, California, USA: Association for Computing Machinery, 1991, pp. 63–74. doi: 10.1145/106972.106981.
- [2] J. M. Domingos, T. Rocha, N. Neves, N. Roma, P. Tomás, and L. Sousa, "Supporting RISC-V Performance Counters Through Linux Performance Analysis Tools," in *2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2023, pp. 94–101. doi: 10.1109/ASAP57973.2023.00027.
- [3] U. Drepper, "What Every Programmer Should Know About Memory," 2007, [Online]. Available: <https://www.akkadia.org/drepper/cpumemory.pdf>
- [4] H. Brais, R. Kalayappan, and P. R. Panda, "A Survey of Cache Simulators," *ACM Comput. Surv.*, vol. 53, no. 1, Feb. 2020, doi: 10.1145/3372393.
- [5] N. L. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011, doi: 10.1145/2024716.2024718.
- [6] J. Lowe-Power *et al.*, "The gem5 Simulator: Version 20.0+," *CoRR*, 2020, [Online]. Available: <https://arxiv.org/abs/2007.03152>
- [7] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, in ATEC '05. Anaheim, CA: USENIX Association, 2005, p. 41.
- [8] M. Badaroux, J. Dumas, and F. Petrot, "Fast Instruction Cache Simulation is Trickier than You Think," in *Proceedings of the DroneSE and RAPIDO: System Engineering for Constrained Embedded Systems*, in RAPIDO '23. Toulouse, France: Association for Computing Machinery, 2023, pp. 48–53. doi: 10.1145/3579170.3579261.
- [9] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, in PACT '08. Toronto, Ontario, Canada: Association for Computing Machinery, 2008, pp. 72–81. doi: 10.1145/1454115.1454128.
- [10] X. Zhan, Y. Bao, C. Bienia, and K. Li, "PARSEC3.0: A Multi-core Benchmark Suite with Network Stacks and SPLASH-2X," *SIGARCH Comput. Archit. News*, vol. 44, no. 5, pp. 1–16, Feb. 2017, doi: 10.1145/3053277.3053279.
- [11] D. Patterson *et al.*, "Embench IOT 2.0 and DSP 1.0: Modern Embedded Computing Benchmarks," *Computer*, vol. 58, no. 5, pp. 37–47, 2025, doi: 10.1109/MC.2024.3511352.
- [12] P. J. Fleming and J. J. Wallace, "How not to lie with statistics: the correct way to summarize benchmark results," *Commun. ACM*, vol. 29, no. 3, pp. 218–221, Mar. 1986, doi: 10.1145/5666.5673.
- [13] Z. Chen and Y. Yang, "Assessing forecast accuracy measures," p. , 2004.
- [14] "ARM Developer Suite Developer Guide : The branch and exchange instruction." 2001. [Online]. Available: <https://developer.arm.com/documentation/dui0056/d/interworking-arm-and-thumb/assembly-language-interworking/the-branch-and-exchange-instruction>
- [15] M. Badaroux, "Dynamic Binary Translation speed and accuracy trade-offs : investigating parallel scalability and cache simulation," 2024. [Online]. Available: <https://theses.hal.science/tel-04720001>