Fast Instruction Cache Simulation is Trickier than You Think

Marie Badaroux, Julie Dumas, Frédéric Pétrot

Fast Instruction Cache Simulation is Trickier than You Think

Marie Badaroux
Univ. Grenoble Alpes, CNRS, Grenoble INP†, TIMA
38000 Grenoble, France
marie.badaroux@univ-grenoble-alpes.fr

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

Frédéric Pétrot
Univ. Grenoble Alpes, CNRS, Grenoble INP†, TIMA
38000 Grenoble, France
frederic.petrot@univ-grenoble-alpes.fr

ABSTRACT

Given the performances it achieves, dynamic binary translation is the most compelling simulation approach for cross-emulation of software centric systems. This speed comes at a cost: simulation is purely functional. Modeling instruction caches by instrumenting each target instruction is feasible, but severely degrades performances. As the translation occurs per target instruction block, we propose to model instruction caches at that granularity. This raises a few issues that we detail and mitigate. We implement this solution in the QEMU dynamic binary translation engine, which brings up an interesting problem inherent to this simulation strategy. Using as test vehicle a multicore RISC-V based platform, we show that a proper model can be nearly as accurate as an instruction accurate model. On the PolyBench/C and PARSEC benchmarks, our model slows down simulation by a factor of 2 to 10 compared to vanilla QEMU. Although not negligible, this is to be balanced with the factor of 20 to 60 for the instruction accurate approach.

ACM Reference Format:
Marie Badaroux, Julie Dumas, and Frédéric Pétrot. 2023. Fast Instruction Cache Simulation is Trickier than You Think. In *15th Workshop on Rapid Simulation and Performance Evaluation for Design Optimization: Methods and Tools, January 16-18, 2023, Toulouse, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3579170.3579261

1 INTRODUCTION

Hardware/Software co-simulation is a technology that is very useful during the design, implementation and even actual use of processor centric systems. It can be implemented in many different ways [14, 22], as does the simulator responsible for the execution of the software parts of the system. For this latter task, Dynamic Binary Translation (DBT) has proven to be very efficient and scalable. Its simulation strategy is based on the following consideration: a sequence of non-branch instructions ended by a branch can be executed atomically. Therefore, by translating a sequence once and reusing it for its many executions, the translation time is amortized, and the speed of simulation dictated mainly by the speed of execution of the translated code.

When it comes to evaluating performances, DBT is challenging. Speed is due to the execution of a lot of translated instructions out of the control of the simulation environment, and adding instrumentation will rapidly incur large slowdowns.

In this work, we take benefit from the block per block translation nature of the DBT to propose a fast yet accurate instruction cache model. We study the sources of inaccuracy of our model and detail the choices we made to mitigate them.

We assess simulation performance in terms of accuracy and simulation speed using the QEMU [4] dynamic binary translator emulating the RISC-V instruction set architecture (ISA) running the PolyBench/C [23] programs as bare-metal, and the multi-threaded PARSEC benchmark [5] in *user-mode*. We use as baseline for comparison the current instruction level instrumentation available in vanilla QEMU. We show that performing a fair comparison is not simple, and propose a way to circumvent this issue.

The paper is organized as follows. We first outline in Section 2 the DBT process. Section 3 gives an overview of the most relevant related works on instruction cache simulation and instrumentation. Section 4 details the modeling approach we propose and its limitations and Section 5 details the implementation in QEMU. Section 6 presents the environment we put to work to run our experiments, and the results we obtained. Finally, we conclude in Section 7.

2 DYNAMIC BINARY TRANSLATION PRIMER

Figure 1 outlines the DBT process. In the frontend, the target instructions are fetched one at a time, decoded, translated into micro-operations (μops), and sequentially added into a buffer. If the current instruction is a branch, we enter the backend in which the μops in the buffer are translated into optimized host instructions, and put in a Translation Block (or TB). A prologue is added in front of the instructions for housekeeping, as is an epilogue to give back control to the simulation environment. The identifier of the sequence of code is the value of the program counter (pc) of the first instruction of the sequence. The TB is then stored into a translation cache, and immediately executed. Once TB execution finished, the control is returned to the simulator that seeks the next TB to execute, identified by the address of the instruction that is the target of the branch ending the previous TB.

The notion of translation block is very similar to the notion of basic-block coined by [1] in the static compilation context, but still, it has some differences. Indeed, the dynamic nature of the compilation makes it possible to have several TB representing the exact same sequence of target instructions, if the sequence is the target of a goto that jumps into a larger sequence. Also, translation occurs within a known processor environment, including in particular the machine state, *e.g.* kernel or user-mode, page-table settings, etc. This means in particular that the addresses of the target instructions, be they physical or virtual, are statically known during TB

† Institute of Engineering Univ. Grenoble Alpes.

RAPIDO'23, January 16-18, 2023, Toulouse, France

ACM ISBN 979-8-4007-0045-3/16/01…$15.00
https://doi.org/10.1145/3579170.3579261

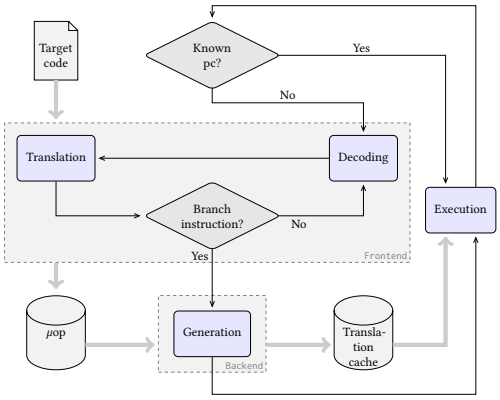

Figure 1: Outline of the dynamic binary translation process.

generation. Additionally, TBs need to be deleted and retranslated under certain circumstances, such as when dynamically changing an instruction (when adding a breakpoint, patching a dynamically called function address, *etc*), or when the translation cache is full. Last, but not least, it might happen that a TB is only partially executed, when an exception is raised upon execution of an inner target instruction. For instance, a memory access might trigger a page-fault that will immediately end the current TB to fetch the exception handler.

3 RELATED WORKS

A recent survey on cache simulators is available in [6]. As explained in this necessarily non-exhaustive survey, about 30 years ago cache simulators were mainly studied for research purposes and were the alternative for when the corresponding hardware was not available. The tool representative of this period is Dinero, whose version IV [10] is the latest. Nowadays, these simulators are also used to help optimize software stacks, including parallel ones.

Instrumentation is in important mechanism to analyse software but it can be challenging to implement. This topic has been addressed by [17, 18] that achieve good performance. The Dynamic Binary Instrumentation framework Valgrind [21] proposes a more complete instrumentation for heavyweight Dynamic Binary Analysis. With the module Cachegrind, the analysis tool Valgrind is able to simulate classical cache hierarchies [25]. Cross-instrumentation, although based on the same principles, is less common [9, 13].

Adding representation of new structures in simulators can be challenging as it can lead to a non negligible execution time overhead. More than 20 years ago, an extension to the instruction set simulator SimICS was proposed by [19] to accurately simulate an instruction cache. Even if the resulting simulation was slowed down, it was deemed acceptable given the accuracy gained. In 2010, [8] presented a high-level instruction cache model. Their approach applied to co-simulation did not degrade simulation speed too much.

Thanks to its execution speed, DBT stands as one of the most used solution for functional high level simulation. However, it lacks microarchitectural details. Works like [3] produce a DBT-based simulation with addition of architectural details and report a good trade-off between speed and precision.

Works to include cache simulation in QEMU have already been initiated long ago [12, 24], but were quite intrusive and architecture dependent. [7] also extended the dynamic binary translator QEMU and claim time-accurate multiprocessor simulation. With the recent introduction of plugins into QEMU, cache simulation can be done non-intrusively. It is already available upstream [20] with a simple cache model per virtual CPU (i.e. emulated target CPU or vCPU in short). Our goal is to propose an instruction cache model specific to the DBT mechanism and to show that the overhead it incurs can be significantly reduced.

4 INSTRUCTION CACHE MODELING

A high-level instruction cache model generally implements only the directory holding the tags and a bit indicating if the tag at a given index is valid or not. The index of the line[1] at which an address is stored in the directory, indistinctive of the exact cache geometry, is computed as a combination of the upper address bits (tag) and/or middle address bit (index). The lower bits of the address indicate the exact instruction to fetch within the line, but they are not relevant since the line is either valid as a whole, or invalid as a whole. Given the fact that TB translation occurs with a known virtual to physical address translation context, virtually addressed, physically addressed, or virtually indexed physically tagged caches can be simulated.

4.1 Initial Intuition

Given the TB per TB execution principle at work in DBT, we have the warranty that once we have entered a TB, all subsequent instructions of the TB are at consecutive (either 2 or 4 bytes away in the RISC-V case) addresses. So, we need to check when entering the TB whether or not the first instruction misses, as we have no prior knowledge. But from then on, we are sure that all following instructions that share the same index will hit. Then, when the index changes, we might have a miss, but again, for the instructions that follow, we are sure they will hit. So, because addresses are consecutive inside a TB, we can a priori know which instructions will hit, and which might miss. So we know at TB creation time that it is sufficient to check for this subset of instructions dynamically. Figure 2 illustrates this principle as an example.

```
0x800fa7bc:  1141      addi  sp,sp,-16   ← possible miss
0x800fa7be:  e022      sd    s0,0(sp)    ← hit!
0x800fa7c0:  e406      sd    ra,8(sp)    ← possible miss
0x800fa7c2:  0800      addi  s0,sp,16    ← hit!
0x800fa7c4:  00dbc797  auipc a5,14401536 ← hit!
0x800fa7c8:  2347a783  lw    a5,564(a5)  ← hit!
0x800fa7cc:  eb95      bnez  a5,52       ← hit!
```

Figure 2: Example of static hit/miss decision within a TB.

Without loss of generality, we assume a 16-byte cache line, which means that the line base addresses have their 4 least significant bits zeroed, and the RISC-V ISA. We jump into this TB at address `0x800fa7bc`, which lays in the middle of a line. We have to check if

[1]To avoid ambiguity, we consistently use *line* for consecutive cache elements (a.k.a block) and *block* for what refers to translated target instructions.

reading the instruction at this address misses, but we know ahead of time that the next address, `0x800fa7be`, will hit. The next instruction is at `0x800fa7c0`, which might be a miss as it stands on a new line, so we must check it. However, the 4 following instructions will for sure hit, since they belong to the same line. Overall, on this TB, we have to run the cache simulator for 2 addresses, while 7 instructions are executed. As a result, we considerably reduce the overhead of doing cache simulation.

4.2 Error in Counting Instructions and How to Mitigate it

Unfortunately, the occurrence of exceptions is like a grain of sand seizing up this well oiled mechanism. When we enter a TB, we make the assumption that *all* instructions it contains will be executed atomically, and therefore run our model on all addresses that might miss it contains. Alas, when an exception occurs, what is left of the TB beyond the faulty instruction is not executed, and control is handed over to the simulation environment to fetch the instructions of the exception handler. Then on return from the handler, since the `pc` of the instruction that was following the access was not known, what was left of the TB is retranslated into a new smaller TB, and accounted for by our model. Figure 3 illustrates this with actual page-faults taking place during Linux boot, once the kernel page table has been set-up. We see that there are three re-translations, as the first two TBs have only been partially executed, and that the two last TBs are subsets of the first one. In that situation, we ran our cache simulator on 6 non actually executed instructions in the first TB, and on 3 on the second TB. Note that the faulty instruction is considered twice, once in the original TB, and once in the retranslated TB, but this is expected as the cache might have been trashed by the exception handler. Instructions such as `wfi` (that sleeps waiting for an interrupt) or `pause` (that yields back the processor) induce the same behavior, but they are easy to handle. Indeed, they occur so rarely that forcing them to end a TB induces an insignificant slowdown.

On the contrary, memory accesses occur very often and might represent from 30% to 50% of the instruction mix in some actual workloads [11, 15]. Using the same strategy as previously might incur too much of a slowdown for a negligible accuracy gain, but this needs to be checked, and it will be in Section 6.

4.3 Dependency on Simulator Runtime

Another, deeply annoying, unexpected behavior is the dependencies of the flow of executed target instructions on the time it takes to perform cache simulation for programs running on top of Linux[2]. We discovered this seemingly odd behavior while testing several cache implementations: to our amazement, the faster the simulator, the lower the number of executed instructions for a given program. To the best of our knowledge, that effect has not yet been reported in the literature. This makes the evaluations obtained in previous works (we specifically refer here to the ones using QEMU, but we see no reason why similar tools would behave differently), *e.g.* [12, 16, 24], questionable, although no wrong per se, since the executed instructions produce a valid behavior.

```
# Insns in translation block
0x7f1ffbe5692c :   auipc     a5,237568
0x7f1ffbe56930 :   ld        a5,-612(a5)
0x7f1ffbe56934 :   sb        s0,0(a5)
0x7f1ffbe56938 :   ld        ra,8(sp)          ┐
0x7f1ffbe5693a :   auipc     a5,270336         │
0x7f1ffbe5693e :   sb        s0,1470(a5)       │ 6 instructions
0x7f1ffbe56942 :   ld        s0,0(sp)          │
0x7f1ffbe56944 :   addi      sp,sp,16          │
0x7f1ffbe56946 :   ret                         ┘
# Executed insns until page fault
0x7f1ffbe5692c :   auipc     a5,237568
0x7f1ffbe56930 :   ld        a5,-612(a5)
0x7f1ffbe56934 :   sb        s0,0(a5)     ⟵ first page-fault
# New translation block after return from handler
0x7f1ffbe56934 :   sb        s0,0(a5)
0x7f1ffbe56938 :   ld        ra,8(sp)
0x7f1ffbe5693a :   auipc     a5,270336
0x7f1ffbe5693e :   sb        s0,1470(a5)
0x7f1ffbe56942 :   ld        s0,0(sp)          ┐
0x7f1ffbe56944 :   addi      sp,sp,16          │ 3 instructions
0x7f1ffbe56946 :   ret                         ┘
# Executed insns until new page fault
0x7f1ffbe56934 :   sb        s0,0(a5)
0x7f1ffbe56938 :   ld        ra,8(sp)
0x7f1ffbe5693a :   auipc     a5,270336
0x7f1ffbe5693e :   sb        s0,1470(a5) ⟵ second page-fault
# Again, new translation block after return from handler
0x7f1ffbe5693e :   sb        s0,1470(a5)
0x7f1ffbe56942 :   ld        s0,0(sp)
0x7f1ffbe56944 :   addi      sp,sp,16
0x7f1ffbe56946 :   ret
# Executed insns until branch, no page-fault on ld
0x7f1ffbe5693e :   sb        s0,1470(a5)
0x7f1ffbe56942 :   ld        s0,0(sp)
0x7f1ffbe56944 :   addi      sp,sp,16
0x7f1ffbe56946 :   ret
```

Figure 3: Stopped TB execution due to a page-fault.

We investigated to understand the reasons behind this behavior, and we discovered they are due to repeated occurrences of timer interrupts because the simulator uses the host real-time clock (obtained, e.g. using `rdtsc` on x86 hosts) to trigger alarms. In this situation, the timer interruption is raised when a given wall-clock Δt has elapsed, which finally ends up calling the target Linux `update_cfs_group` scheduling function over and over. To exemplify the behavior, we have added in the cache model a delay using a `for` loop iterating n times, which leads to the results given Figure 4. This can be mitigated by forcing QEMU to use a clock of its own[3] in which case only 1830 irqs are raised, for any added delay. However, in that case it cannot run vCPUs in parallel, which greatly hinders performance. As we are aiming to show that our cache model behaves as expected and compare its execution speed to an existing

[2]Bare metal programs do not suffer from this.

[3]Using the `-icount shift=2,sleep=on` option.

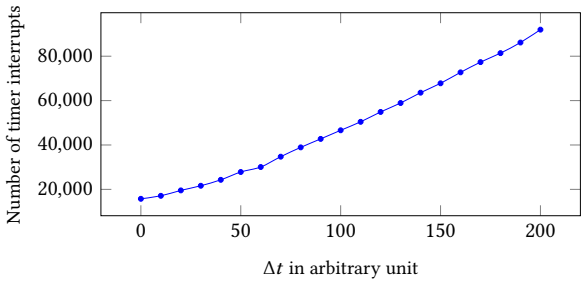

Figure 4: Number of raised timer interrupts as a function of an arbitrary cache model simulation delay.

model or to vanilla QEMU, we must have the exact same instructions executed to be fair. To that end, we will use either bare metal software that does not program alarms or QEMU user-mode, both taking benefit of the host parallelism while having no dependency on simulation duration.

5 IMPLEMENTATION

5.1 QEMU TCG Plugins

The Tiny Code Generator (TCG) plugin is a feature introduced in QEMU version 4.2 [9]. It provides an API to write plugins in order to facilitate code instrumentation. Information is given through the plugin API at translation and execution times. It can be retrieved each time a block is translated or/and executed, or for every instruction and memory access that is executed. By instrumenting all instructions and memory accesses executed by the target through the TCG plugins, we have access to all the information we need to simulate a cache.

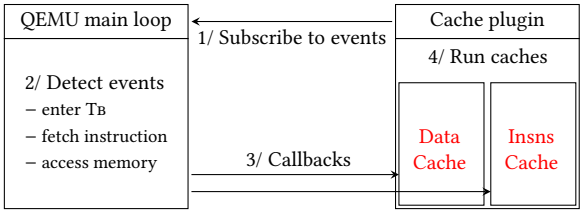

Figure 5: Simplified representation of the QEMU TCG plugins mechanism.

Figure 5 shows the simplified plugin mechanism of QEMU's TCG. First, we have to indicate to which events proposed by the TCG plugin API we want to subscribe. It can be an instruction or a memory access execution or a translation block translation/execution. Then, during QEMU execution, each time the event occurs the main loop will send information to the plugin thanks to callback functions. Finally, inside the plugin, we can add our own code to do what we want. For this work, we do not consider the data-cache and solely focus on the instruction cache. TCG plugins are independent from the simulated architecture and thus can be used with all the different targets QEMU supports. In our case, we decided to work only with RISC-V but the approach we propose can be applied to all the other architectures as well.

5.2 Cache Simulation at Tb Granularity

The TCG plugin API is limited but offers the possibility to retrieve the target instructions belonging to a Tb during translation. During that stage, we have two options.

The first one, implemented by the existing instruction cache plugin since 2021 [20], is to register a callback function for every instruction of the Tb. The function will be called right before instruction execution, once all its operand values known. Being at instruction granularity, the cache model is invoked each time an instruction is executed, which results in an important simulation time overhead.

The second one is to register a callback each time a Tb is executed, which fits well the DBT principles, this is the option that we chose as introduced in Section 4. During the translation stage, thanks to the API of the TCG Plugins, we can parse the Tb and have access to the instructions it contains. Our plugin records in an array the addresses of the instructions that might produce a miss, and this array is given as argument to the function of the API that registers a callback for the Tbs. Then, just before a Tb is executed, the callback traverses the array and performs cache simulation on the sequence of addresses at once.

Using an array to store a subset of instructions requires dynamic memory allocation. However, thanks to the DBT mechanism, the Tbs are put in a cache to be reused. Even if it is application dependant, the percentage of Tb reuse is really high, so the per Tb memory allocation is negligible.

6 EXPERIMENTS

To evaluate our solution, we use two benchmarking suites: PolyBench/C for uniprocessor, and PARSEC for multiprocessors. The PolyBench/C suite is run with the MEDIUM inputs and the PARSEC suite with the LARGE inputs. We decided to work with the RISC-V target. We used QEMU in two different modes: *user-mode*, in which system calls and signals are handled by the host OS, and *full-system*, which is a Linux-capable target.

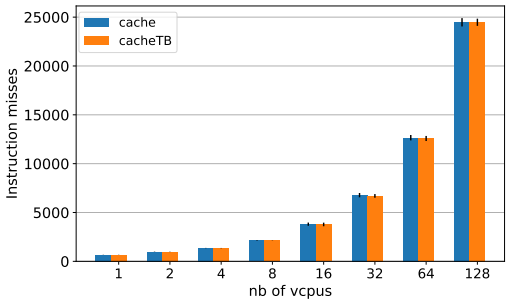

Figure 6: Number of instruction misses for lu_cb (log scale on x-axis).

QEMU, Linux and the machine we run on are not time deterministic. We therefore run our experiments 20 times to measure performances, which, by the virtue of the central limit theorem and given the standard deviation we obtain, gives a high confidence in the computed execution time mean.

In the following figures, `cache` refers to QEMU existing cache plugin, `cacheTB` refers to QEMU with a plugin that implements our solution and `vanilla` refers to QEMU without any plugin.

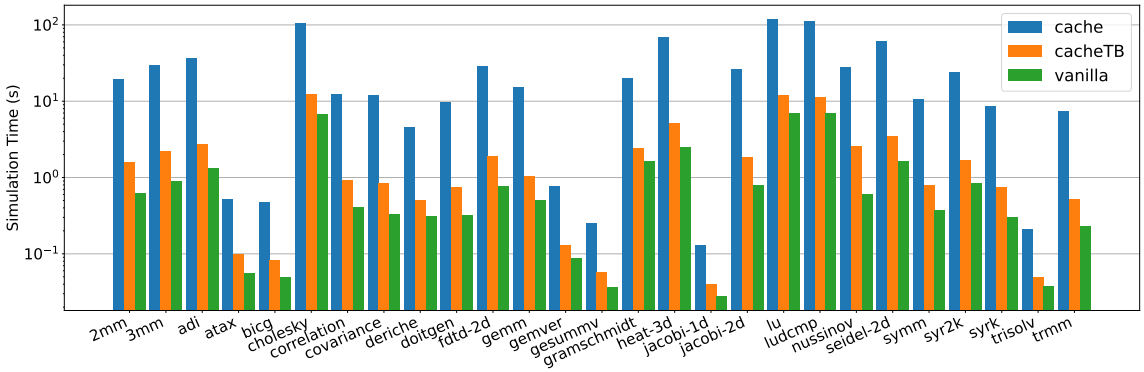

Figure 7: Simulation time of the PolyBench/C programs (log scale on y-axis).

6.1 Error due to exceptions in TBs

Because early internal TBs exits can occur, we need to make sure the error it incurs is negligible. To that aim, we choose 2 programs that we run on a single vCPU: a boot of Linux that finishes just before printing the login prompt, and the LU decomposition of a 2048 × 2048 matrix from the parsec. We measure the total number of executed TBs, the number of internal TBs exits, the total number of instructions and the number of instructions that have been accounted for while they shouldn't. This is reported in Table 1. The average number of target instructions per TB is 5.51 for Linux boot, and 16.61 for LU, which outlines the differences between both workloads. Although the error in number of instructions is negligible, we made a change in QEMU in which each memory access ends a TB. For Linux boot, the number of executed TBs almost quadruples, to 124,097,849, while the total simulation time increases by 50%. This is not a surprise, as in that workload, 44% of the instructions are memory accesses.

Table 1: Measure of the error due to early TB exits.

	Nb of TBs executed	Nb of early TBs exits	Nb of executed insns	Wrongly counted insns
Boot	35,704,017	254	196,831,388	669
LU	989,522,360	10,447	16,439,546,310	6098

In conclusion, our approach to consider all instructions that belong to a TB as executed makes sense. It is not useful to mitigate it, since it costs a lot in simulation time while having zero impact on the cache statistics.

6.2 Statistics validation using QEMU existing cache plugin

We report in this section how we validated the statistics produced by our plugin. Although we thoroughly tested the behavior of our cache model against QEMU's current one, all the experiments were done with the following arbitrary instruction cache configuration: 8-way, 32-set, 64 bytes per line and LRU.

6.2.1 Unicore: PolyBench/C Suite. We used the PolyBench/C suite on bare-metal to compare the statistics of our `cacheTB` plugin with the existing `cache` plugin. We obtained the exact same number of instructions and instruction misses with both plugins for the whole benchmark. Thus we have been able to confirm the good behavior of our model in monoprocessor.

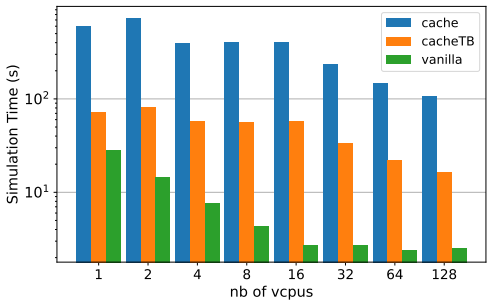

Figure 8: Simulation time of lu_cb (log-log scale).

6.2.2 Multicore: PARSEC Suite. In order to validate our cache model without the issues raised in Section 4 when using Linux, we run the PARSEC suite in QEMU *user-mode*. Figure 6 shows the number of instruction misses for the number of virtual CPUs from 1 to 128 by integer power of 2 steps for the *lu_cb* PARSEC benchmark. The vertical black lines on the figure represent the range of statistics we did for 20 executions. We obtained for each of the vCPUs a mean of the statistics (number of instructions and number of instruction misses) that are really close from the ones of the `cache` plugin and the values are stable across the 20 executions. For all the PARSEC benchmarks that we used, we observed similar histograms than the one of *lu_cb*.

6.3 Simulation time: `vanilla` vs `cache` vs `cacheTB`

In this section, we compare the simulation time of QEMU `cache` plugin and our own `cacheTB` plugin. Figure 7 compares the simulation time of the benchmarks of the PolyBench/C suite in bare metal. Figure 8 compares the simulation time of the PARSEC program *lu_cb* run in *user-mode* with 1 to 128 virtual CPUs. Instead of showing the simulation time histograms for all PARSEC programs that we used and because they have all the same shape, we represent Figure 9 the simulation time using 128 virtual CPUs only for all

Table 2: Mean simulation time ratios.

	PolyBench/C	PARSEC
Speedup **cache** to **cacheTB**	10.87	7.18
Overhead **cache** to **vanilla**	23.67	59.85
Overhead **cacheTB** to **vanilla**	2.07	10.16

programs (on a 2 GHz 64-core AMD EPYC 7702P PowerEdge R6515 server), outlining the scalability of our approach, QEMU in itself being fairly scalable [2].

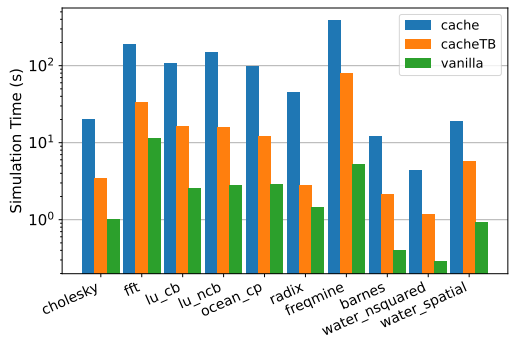

Figure 9: Simulation time of the PARSEC programs on 128 vCPUs (log scale on y-axis).

Table 2 summarizes all the different mean speedup/overhead ratios. According to the different types of experiments, our plugin is 7 to 10 times faster than the cache plugin. Regarding the overhead of using a plugin compared to QEMU vanilla, the cache plugin degrades much more the simulation time than our plugin (up to 60 times slower than QEMU vanilla with the PARSEC Suite compared to only 10 times slower with our model).

7 CONCLUSION

Although adding instrumentation in functional simulation will for sure degrade simulation time, taking benefit of the per block nature of the translation in DBT allows to define a strategy suited for instructions caches that minimizes the performance overhead. We raised two issues, one due to the DBT itself that we could mitigate, the other linked to time handling that we circumvent, but didn't solve per se. We implemented our approach in QEMU and validated against its existing instruction accurate cache model. For all the experiments we did, we obtained identical statistics, but at far better performances. Overall, fast instruction cache simulation isn't that easy, and even though doing it at scale for full-system is possible, asserting the validity of the metrics that are produced is still a work in progress.

ACKNOWLEDGMENTS

We would like to thank the partners of the ANR Rakes project and acknowledge the financial support of the French Agence Nationale de la Recherche (ANR-18-CE25-0017, https://anr.fr/Project-ANR-18-CE25-0017).

REFERENCES

[1] John W. Backus, Robert J. Beeber, Sheldon Best, Richard Goldberg, Lois M. Haibt, Harlan L. Herrick, Robert A. Nelson, David Sayre, Peter B. Sheridan, Harold Stern, Irving Ziller, R. A. Hughes, and R. Nutt. 1957. The FORTRAN automatic coding system. In *Proceedings of the Western joint computer conference: Techniques for reliability*. 188–198.

[2] Marie Badaroux, Saverio Miroddi, and Frédéric Pétrot. 2021. To Pin or Not to Pin: Asserting the Scalability of QEMU Parallel Implementation. In *24th Euromicro Conference on Digital System Design*. IEEE, 238–245.

[3] Igor Böhm, Björn Franke, and Nigel Topham. 2010. Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator. In *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. IEEE, 1–10.

[4] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*. USENIX, 41–46.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. IEEE, 72–81.

[6] Hadi Brais, Rajshekar Kalayappan, and Preeti Panda. 2020. A Survey of Cache Simulators. *ACM Computing Surveys (CSUR)* 53 (02 2020), 1–32.

[7] Humberto Carvalho, Geoffrey Nelissen, and Pavel Zaykov. 2020. mcQEMU: Time-Accurate Simulation of Multi-core platforms using QEMU. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*. IEEE, 81–88.

[8] Juan Castillo, Hector Posadas, Eugenio Villar, and Marcos Martinez. 2010. Fast Instruction Cache Modeling for Approximate Timed HW/SW Co-Simulation. In *Proceedings of the 20th Symposium on Great Lakes Symposium on VLSI*. IEEE, 191–196.

[9] Emilio Cota and Luca Carloni. 2019. Cross-ISA machine instrumentation using fast and scalable dynamic binary translation. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, 74–87.

[10] Jan Edler and Mark D Hill. 1998. Dinero IV Trace-Driven Uniprocessor Cache Simulator. https://pages.cs.wisc.edu/~markhill/DineroIV/.

[11] Antoine Faravelon, Olivier Gruber, and Frédéric Pétrot. 2021. *Removing Load/Store Helpers in Dynamic Binary Translation*. John Wiley & Sons, Ltd, Chapter 7, 133–160.

[12] Marius Gligor, Nicolas Fournel, and Frédéric Pétrot. 2009. Using Binary Translation in Event Driven Simulation for Fast and Flexible MPSoC Simulation. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 71–80.

[13] Christophe Guillon. 2011. Program instrumentation with qemu. In *1st International QEMU Users' Forum*, W. Mueller and F. Pétrot (Eds.), Vol. 1. 15–18.

[14] Fatma Jebali, Oumaima Matoussi, Arief Wicaksana, Amir Charif, and Lilia Zaourar. 2022. Decoupling Processor and Memory Hierarchy Simulators for Efficient Design Space Exploration. In *15th Workshop on Rapid Simulation and Performance Evaluation for Design Optimization: Methods and Tools*. ACM, 47–52.

[15] Lizy Kurian John, Vinod Reddy, Paul T. Hulina, and Lee D. Coraor. 1995. Program balance and its impact on high performance RISC architectures. In *First IEEE Symposium on High-Performance Computer Architecture*. IEEE, 370–379.

[16] Shin-haeng Kang, Donghoon Yoo, and Soonhoi Ha. 2016. TQSIM: A fast cycle-approximate processor simulator based on QEMU. *Journal of Systems Architecture* 66 (2016), 33–47.

[17] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[18] Yi-Hong Lyu, Ding-Yong Hong, Tai-Yi Wu, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. 2014. Dbill: An efficient and retargetable dynamic binary instrumentation framework using llvm backend. *Acm Sigplan Notices* 49, 7 (2014), 141–152.

[19] Peter S. Magnusson. 1997. Efficient instruction cache simulation and execution profiling with a threaded-code interpreter. In *Proceedings of the 29th conference on Winter simulation*. IEEE, 1093–1100.

[20] Mahmoud Mandour. 2021. Cache Modelling TCG Plugin. https://www.qemu.org/2021/08/19/tcg-cache-modelling-plugin/.

[21] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *ACM Sigplan notices* 42, 6 (2007), 89–100.

[22] Frédéric Pétrot, Nicolas Fournel, Patrice Gerin, Marius Gligor, Mian Muhammad Hamayun, and Hao Shen. 2010. On MPSoC Software Execution at the Transaction Level. *IEEE Design & Test of Computers* 28, 3 (2010), 2–11.

[23] Louis-Noël Pouchet and Tomofumi Yuki. 2015. Polybench/C 4.1. http://polybench.sourceforge.net.

[24] Tran Van Dung, Ittetsu Taniguchi, and Hiroyuki Tomiyama. 2014. Cache simulation for instruction set simulator QEMU. In *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*. IEEE, 441–446.

[25] Josef Weidendorfer. 2008. Sequential performance analysis with callgrind and kcachegrind. In *Tools for High Performance Computing*. Springer, 93–113.