Dynamic Binary Translation speed and accuracy trade-offs : investigating parallel scalability and cache simulation

Marie Badaroux

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : Techniques de l'Informatique et de la Microélectronique pour l'Architecture des systèmes intégrés

Compromis en termes de vitesse et de précision de la Traduction Binaire Dynamique : étude du passage à l'échelle de la version parallèle et d'une simulation de cache

Dynamic Binary Translation speed and accuracy trade-offs: investigating parallel scalability and cache simulation

Présentée par :

Marie BADAROUX

Direction de thèse :

Frédéric PETROT PROFESSEUR DES UNIVERSITES, GRENOBLE INP	Directeur de thèse
Julie DUMAS MAITRE DE CONFERENCES, GRENOBLE INP	Co-encadrante de thèse

Rapporteurs :

ERVEN ROHOU
DIRECTEUR DE RECHERCHE, CENTRE INRIA DE L'UNIVERSITE DE RENNES
TANGUY RISSET
PROFESSEUR DES UNIVERSITES, INSA LYON

Thèse soutenue publiquement le **12 mars 2024**, devant le jury composé de :

MARIE-LAURE POTET, PROFESSEURE DES UNIVERSITES, GRENOBLE INP	Présidente
FREDERIC PETROT, PROFESSEUR DES UNIVERSITES, GRENOBLE INP	Directeur de thèse
ERVEN ROHOU, DIRECTEUR DE RECHERCHE, CENTRE INRIA DE L'UNIVERSITE DE RENNES	Rapporteur
TANGUY RISSET, PROFESSEUR DES UNIVERSITES, INSA LYON	Rapporteur
HENRI-PIERRE CHARLES, DIRECTEUR DE RECHERCHE, CEA CENTRE DE GRENOBLE	Examinateur
KEVIN MARTIN, MAITRE DE CONFERENCES HDR, UNIVERSITE BRETAGNE SUD - LORIENT VANNES	Examinateur

Invités :

JULIE DUMAS
MAITRE DE CONFERENCES, GRENOBLE INP

Remerciements

Je souhaite tout d'abord remercier mes encadrants de thèse de m'avoir suivie pendant ces trois années. En premier merci à Frédéric de m'avoir proposé ce sujet de thèse dans la continuité de mon projet de master. Je ne le remercierai jamais assez pour sa disponibilité et son implication tout au long de cette thèse malgré son emploi du temps très chargé. Je remercie Julie qui s'est greffée à l'encadrement de thèse à la fin de ma première année. Merci pour tout le temps passé à me remonter le moral et à m'écouter me plaindre quand les choses n'avançaient pas. Un grand merci pour tout ce qu'elle a fait pour moi que ce soit sur le plan professionnel ou personnel.

Je remercie ensuite les membres de mon jury. Merci à Marie-Laure Potet d'avoir présidé ce jury. Merci à Erven Rohou et Tanguy Risset d'avoir rapporté sur ces travaux de thèse. Merci à Henri-Pierre Charles pour son rôle d'examinateur et merci à Kévin Martin pour son rôle d'examinateur mais également de m'avoir suivie pendant ces trois ans en faisant parti de mon Comité de Suivi Individuel.

Je remercie toutes les personnes de l'équipe SLS. Merci à l'ensemble des permanents, Arthur, Olivier, Liliana, Laurence, Frédéric R. qui individuellement m'ont conseillé et aidé à traverser cette aventure qu'est la thèse. Un grand merci aux doctorants de l'équipe qui sont maintenant devenus des amis plus que de simples collègues. Merci donc à Benjamin, Pierre, Ambre, Chandana et un merci particulier à Nathan pour ses origamis.

Je tiens également à remercier les membres des équipes pédagogiques de l'Ensimag avec qui j'ai pu travailler pendant ces années. Un merci particulier à François, Matthieu et Olivier pour leurs conseils et à Julie d'avoir cru en moi depuis le début et qui m'a permis de m'épanouir dans le métier d'enseignant.

Je remercie mes amis proches d'avoir été là dès le début et pour le soutien qu'ils m'ont tous apporté pendant les périodes difficiles. Merci particulièrement à Aude, mon amie d'enfance qui me suit depuis des dizaines d'années.

Je souhaite remercier toute ma famille, sans qui je n'en serais pas là aujourd'hui. Merci Papa et Maman de m'avoir donné le goût de l'enseignement et des sciences. Merci à ma petite sœur Lisa d'avoir quitté le Sud-Ouest pour vivre dans la même ville que sa grande sœur.

Enfin, je remercie la personne qui partage ma vie. Merci Aurélie de m'encourager au quotidien à être qui je suis et merci pour ton soutien sans faille.

Abstract

THE semiconductor industry continues its trend towards the production of increasingly efficient designs. Currently, it does so by making these devices more and more complex through the integration of ”everything” on a chip. However, the time to market and manufacturing costs are non-negligible constraints which make the test and evaluation of such designs particularly challenging. Simulation technologies appear as a solution to let the designers do the evaluation in an appropriate time and cost.

Given the performances it achieves and the high level of abstraction it provides, Dynamic Binary Translation (DBT) is the most compelling simulation approach for cross-simulation of software centric systems. The resulting simulation is, however, purely functional. The emerging trend around multi and many-core systems with up to hundreds of cores impacts the simulation mechanisms to make them take advantages of the host architecture and DBT is not an exception to the rule.

Improving the DBT mechanism raises the question of how to keep a good balance between speed and accuracy. On the one hand, working on increasing the DBT speed positively affects the balance. On the other hand, adding modeling for new architectural features in the simulation calls the balance into question, as it degrades the performance. Thus, to stay in line with the principles at work when developing DBT, it is preferable to do so by limiting the overhead induced by the new features.

The first contribution of this thesis aims at increasing the performance of the parallel simulation by investigating thread affinity of the simulated cores on the physical host cores. The second contribution focuses on a functional cache simulation model that takes benefits of the DBT mechanism and more general solutions to reduce the impact of adding a cache simulation in the simulation. We chose QEMU, the most stable and widely used simulator of its kind, as the DBT engine to implement our contributions.

Résumé

L'INDUSTRIE du semiconducteur continue de tendre vers une production de systèmes de plus en plus efficaces. Ceci est rendu possible par la conception de systèmes de plus en plus complexes qui vient à l'intégration de "tout ce qui est possible" sur une unique puce. Cependant, les délais de commercialisation et les coûts de fabrication de ces systèmes sont des contraintes non négligeables qui rendent le test et l'évaluation particulièrement difficiles. Les technologies de simulation apparaissent comme une solution pour permettre aux concepteurs de réaliser l'évaluation dans des délais raisonnable avec un coût approprié.

Compte tenu des performances qu'elle atteint et du haut niveau d'abstraction qu'elle offre, la Traduction Binaire Dynamique (TBD) est l'approche de simulation la plus convaincante pour la simulation croisée de systèmes centrés sur les logiciels. La simulation qui en résulte est cependant purement fonctionnelle. La tendance émergente autour des systèmes multicœurs qui possèdent jusqu'à des centaines de cœurs impacte les mécanismes de simulation et les pousse à tirer parti de l'architecture de la machine hôte et la TBD ne fait pas exception à la règle.

Améliorer les mécanismes de la TBD soulève la question de comment maintenir un bon équilibre entre vitesse et précision. D'une part, travailler sur l'accélération de la vitesse de simulation affecte positivement cet équilibre. D'autre part, l'ajout de la modélisation de nouvelles fonctionnalités architecturales dans la simulation remet en question cet équilibre, car les performances seront dégradées. Ainsi, pour rester en cohérence avec les principes à l'œuvre lors du développement de la TBD, il est nécessaire d'introduire ces modèles en limitant leur surcoût.

La première contribution de cette thèse vise à augmenter les performances de la simulation parallèle en étudiant l'affinité des processus des cœurs simulés sur les cœurs physiques de la machine hôte. La deuxième contribution se concentre sur un modèle de simulation de cache fonctionnel qui profite du mécanisme de la TBD et des solutions générales pour réduire le surcoût d'ajouter une simulation de cache dans la simulation. Nous avons choisi QEMU, le simulateur le plus stable et le plus utilisé de ce type, comme outil reposant sur la TBD pour mettre en œuvre nos contributions.

Contents

Chapter 1

Introduction

THERE are three "high-level" laws governing computer engineering. The first one, empirical and somehow economical, coined by Gordon Moore in 1965, observes that the chip industry should follow a pace of integrating 2 times more transistors on a chip every other year to optimize its fabrication costs. The original graph describing the law from a few points is presented Figure 1.1. This law still holds, but probably not

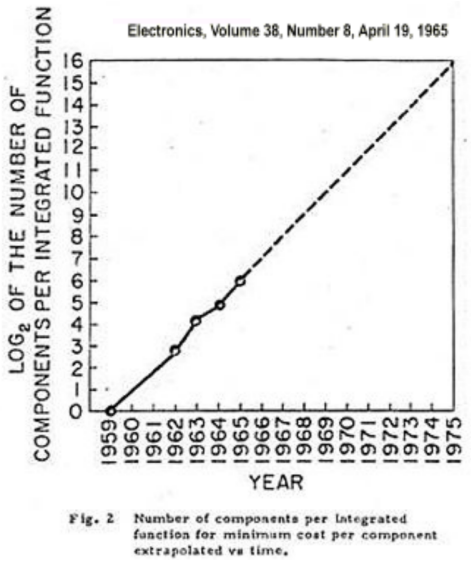


Figure 1.1: Moore's Law

for long given the physics of materials.

The second one, theoretical, is due to Gene Amdahl in 1967. It states that the acceleration that one can expect from distributing its workload on processing elements is limited by the sequential fraction of the workload. It is expressed as $S = \frac{1}{1-f+\frac{f}{s}}$, where S is the speedup that will be gained by parallelizing, f is the fraction of the workload than can be accelerated (*e.g.*, parallelized) by a factor s. Amdahl's law implies that the sequential fraction of the workload should be as small as possible, and that it should be executed as quickly as possible. Multi and many-core systems, thanks to their integration on a single die, help in minimizing overheads and thus mitigating Amdahl's law. It cannot be escaped, though.

The third law, also empirical, is Dennard's law, presented in 1974. It says that scaling down the transistors by half halves the propagation delays while keeping power

consumption identical although transistor density is doubled. This law broke around 2004, as can be seen on Figure 1.2. The frequency, power and number of core curves

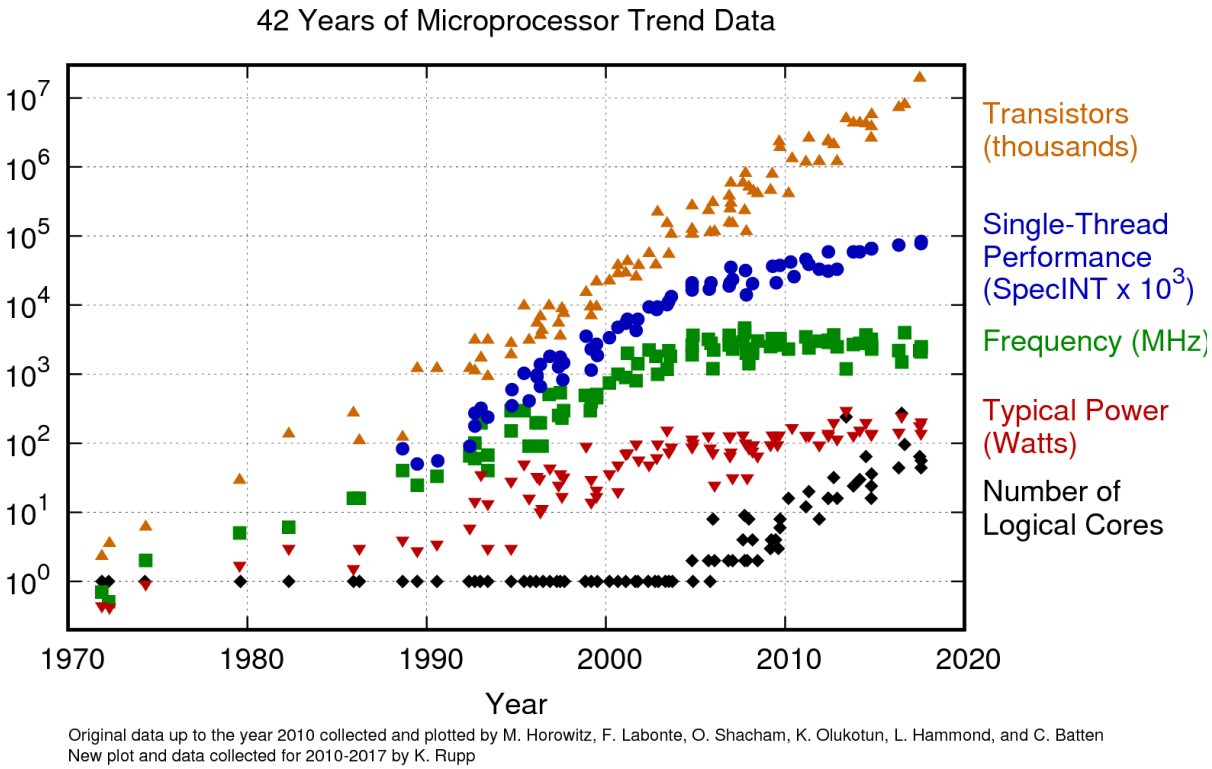


Figure 1.2: Trends in microprocessor integration

have an inflexion point at that date, while the transistor curve (Moore's law in effect) continues to rise steadily. The ability to dissipate power hit a wall at that time, leading to the end of the race for higher and higher frequencies, and to the advent of the multi-core era.

Since then, building more energy efficient chips with more and more cores has been a lively research topic. Designing such chips is however fairly challenging, as manufacturing costs and time to market are stringent constraints. As a tool for testing and evaluating the new designs, simulation technology comes at just the right time for the computer systems architects and designers. Provided these tools supply relevant information, the system designers can do their evaluations in reasonable time at a reasonable cost. Simulation technologies also offer the possibility to implement the software that will run on a specific chip architecture prior to having the actual hardware, which enlarges its usefulness. To follow the trend around multi-core systems over the years, simulation technologies adapted their implementations to take advantage of the multi-core architecture of the host machine and thus reflect more the current state of today systems.

Simulators are divided into a multitude of categories from a very low to a very high level of abstraction. Having a simulator that can simulate precisely all the hardware components and their communication in a system is not often a need. Regarding the simulation of software centric systems, topic to which this thesis belongs, a lot of architectural details can be ignored, and a time-accurate simulation is not necessary. Having the instruction the smallest granularity in the simulation can be sufficient in the vast majority of situations. The resulting simulation is thus purely functional. Historically,

simulators at this level of abstraction are called Instruction Set Simulators and are used for the evaluation of Instruction Set Architecture (ISA) and software making use of it. They provide a good trade-off between accuracy and speed of the simulated system. Dynamic Binary Translation is the most advanced technology in terms of cross-ISA simulation performance, *i.e.*, running guest binary code in a given ISA on a host with a different ISA. One Dynamic Binary Translation engine currently stands out from the crowd: QEMU, and we unsurprisingly worked on this simulator for the practical parts of this thesis.

Improving the Dynamic Binary Translation mechanism, irrespective of in which direction, is of great interest. Given the high level of abstraction it offers, the first direction that interests researchers is increasing its performance. On the other hand, gaining insight on the potential gains in adding new architectural features in the simulation is quite interesting for the system designer. Therefore, the question on how to fulfill a performance/insight trade-off arises. In order to add new features, doing instrumentation seems unavoidable, but it will definitely degrade the global performance.

In this thesis we present contributions that aim at giving accurate simulation results for some architectural features while mitigating the decrease in performances. Two areas of research are studied: the first one investigates thread affinity with the hope to decrease execution time of multi and many-core systems, and the second one focuses on taking advantages of the Dynamic Binary Translation mechanism and general solutions to design a cache model to considerably limits the resulting overhead on execution time.

This manuscript is organized as follows: Chapter 2 presents the motivations of this thesis and what problems we deal with in our domain, Chapter 3 details the works done over the years that bring solutions to these problems, Chapter 4 and Chapter 5 explain our contributions, Chapter 6 presents the experiments and the analysis of our contributions and we conclude and expose the remaining work Chapter 7.

Chapter 2

Problem Statement

THIS chapter describes the problems that are addressed by this thesis. Simulation technologies offer time and cost advantages that are attractive to developers. However, they bring issues on how to fix the limit between speed and accuracy.

First, we present the diverse levels of abstraction present in nowadays systems simulations and why Dynamic Binary Translation (DBT) is one of the best options to do cross simulation when the accurate modeling of hardware details is not necessary. More precisely, we focus on the DBT mechanism and on how it brings a trade-off of speed and accuracy. We then detail the issues when adding new architectural features to the simulation.

Table of contents

2.1 Systems simulation

2.1.1 Processor centric systems

Originally, processor-based systems were composed of a central processing unit that interacts with a storage unit and peripheral units, and datas are exchanged between the units. Nowadays, processors are composed of multiple processing units, named cores, that allow instructions to be executed in parallel which results in an instruction throughput higher than with only one processing unit, and hopefully also higher execution speed. Indeed, due to the frequency limit (*i.e.* end of Dennard's scaling) and as applications are growing in complexity, chip designers were coerced to create processors with multiple cores in them. The first multi-core chip was commercialized by IBM in 2001 with the POWER4 microprocessor, it contained two cores. As a result, it is becoming less common to find mono-core processors and even personal laptops are now equipped with in general 4/8 cores. These kinds of processors are called "multi-core" processors. However, a few cores might still not be enough for some high-performance applications. When we deal with a dozen or more cores, possibly up to a hundred or so, we talk about "many-core" systems, which have a high degree of parallelism. These systems are most of the time arranged in clusters. To illustrate this trend, Figure 2.1 shows the output of the **lstopo** Linux command on a recent server Dell PowerEdge R6515 whose processor is an AMD EPYC 7F72 containing 24 physical cores. One can find in the literature the word `processor`, `core` or `CPU` to refers to the computing unit. There seems to be no universal definition. Therefore, in the context of this thesis, we will use `CPU` to name the smallest unit in the machine and a `core` is the entity that can run multiple hardware threads. Indeed, thanks to the Simultaneous MultiThreading (SMT) technique, multiple threads can run on a mono-core on modern architecture, which improves the parallelism. It gives the Operating System the illusion that there are more computing units available. That is why on Figure 2.1 we say that we have 24 cores and since each core is composed of 2 hardware threads, we have a total of 48 CPUs.

With processor centric systems, the processor can be overwhelmed by the computations. The performances are known to be affected by the data movement across the main memory and the central processing unit, which is also called the Von-Neumann bottleneck. The invention and adoption of cache memories allowed to mitigate this bottleneck. Figure 2.2 presents a simplified vision of the memory hierarchy and how the different kinds of memory are classified. But to meet the needs of applications which are increasingly memory-intensive, memory centric systems have emerged as the solution to replace processor centric systems. The idea is to limit the data movements and with this kind of system, the memory is the center of everything. The memory hierarchy is completely different from the one of the conventional software intensive systems and non-volatile memory seems to be adopted to replace the previous and traditional storage [FKMM15]. In 2015, [FKMM15] announced that at the end of the decade of that time, the computing designers will move towards memory centric architecture. We are two years ahead of this deadline, and lots of work still needs to be done to move to memory centric systems, as current Operating Systems are not built to support such architectures, and the generalization of non-volatile memory is yet to come.

Even if memory centric systems have spread as a research topic, the processor centric systems approach remains attractive and stays the one adopted at scale by the in-

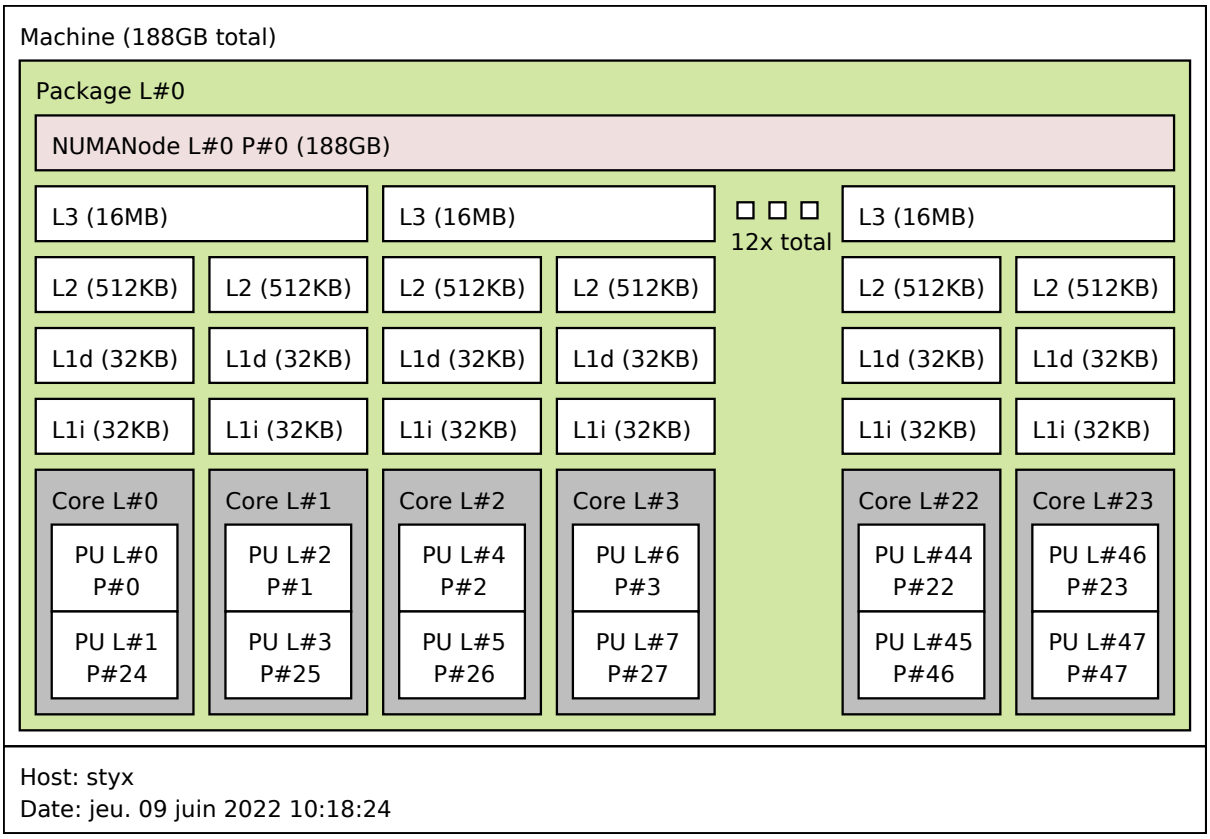


Figure 2.1: lstopo of a Dell PowerEdge R6515

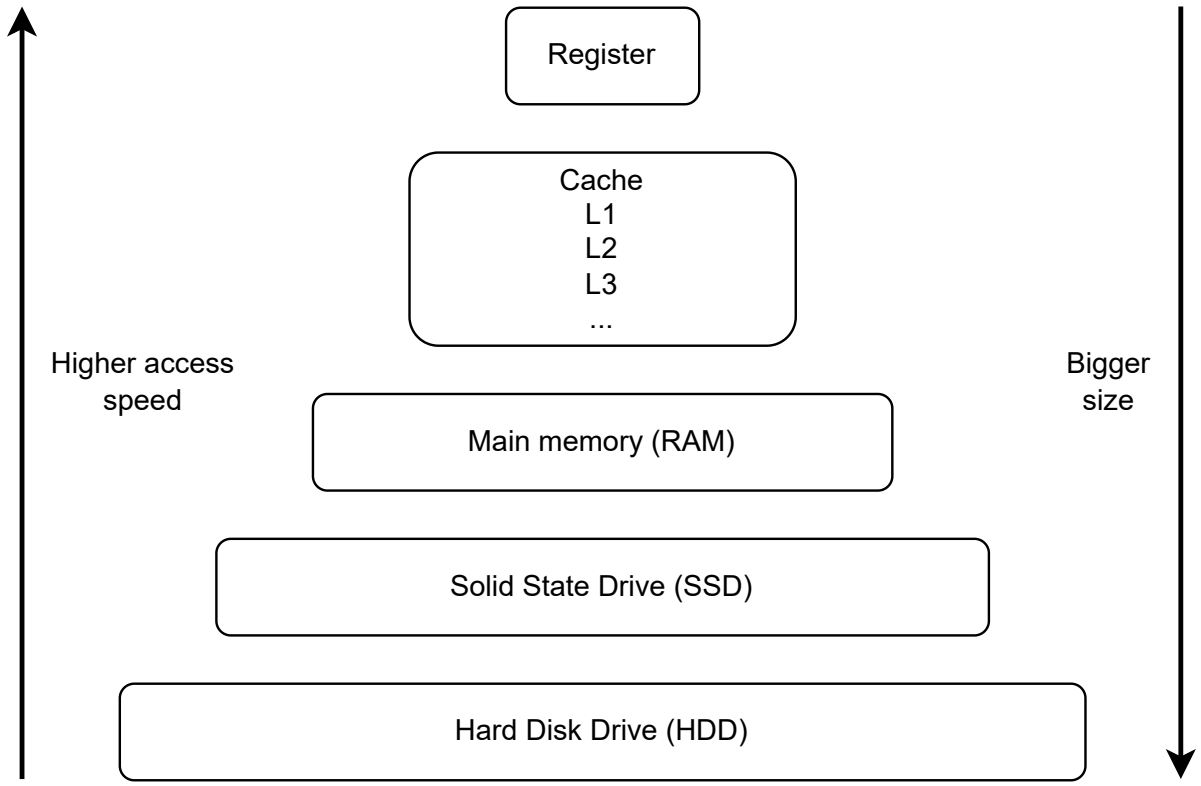


Figure 2.2: Outline of the memory hierarchy

dustry. The trend around multi/many-core processors still creates a variety of academical and industrial researches and many-core systems are currently a booming topic. However, the manufacturing time and cost are such that they greatly restrict what can be done in terms of actual hardware experimentations. As a result, simulators appear to be the solution for testing new design choices when the hardware is not available in a short-term time. In addition, thanks to simulators, developers can debug their software and can test and evaluate, *e.g.*, new Instruction Set Architecture (ISA) extensions, ahead of time. Simulators can also bring new features that the hardware does not provide, for instance the possibility to stop and debug the simulation at any moment, in a non-intrusive way. Finally, simulators can be used at scale, when there are too many developers for the number of devices available, and they can also easily be integrated in continuous integration environments. In summary, full-system simulation is a technology that cannot be avoided.

2.1.2 Different abstraction levels of simulation

Depending on what developers need to test, various kinds of simulators are available. They can be classified into several types with different levels of abstraction and are employed regarding the accuracy/speed trade-off that they provide. Having detailed architectural simulation is a requirement in some research fields. At a very low-level, hardware description abstractions such as the ones provided by Register Transfer Level (RTL) modeling, as implemented by VHDL or Verilog, describe the behavior at such a precision that it can be unambiguously synthesized in hardware. At what can still be called low-level, micro-architectural simulators represent microprocessors with their related components (caches, CPU, memory storage, Translation Lookaside Buffers (TLBs) and so on). Caches are an important hardware component that helps to improve the global execution time of software and cache simulation alone is in itself a major topic. The TLB is a kind of cache memory [CP78], part of the Memory Management Unit (MMU), that stores the recent correspondences, also called translations –hence the name–, between virtual and physical addresses. This is mandatory in any system running user software to warranty process isolation. Being able to simulate the architectural details of these components is important to correctly analyze and evaluate new design choices. However, the emerging complexity of the hardware designs can make the simulation very slow given the details required. Moreover, the pressure performed by the computer and silicon industry to produce and sell quickly new chips forced the developers to improve the performances of simulation techniques while still keeping accurate details of the design.

Since the mid 80's, the gap between the processor and memory performances in processor centric systems has amplified and led to what is currently referenced as the memory wall problem (aka Von-Neumann bottleneck). To reduce this gap, cache memories have a key role to play as they provide higher access speed than the main memory. However, copies need to be managed, and performance of a cache depends on its geometry, size, replacement policy, write policy, and so on. To make the most optimal use of the cache, applications must review their implementation and thus the need of having simulators of only one component such as a cache did emerge. Dinero [EH98] is the historical reference simulator. This tool provides a simulation based on mono-core traces that produces simple statistics such as the number of hit/miss. But this simulator was quickly overtaken by the emerging multi-core processors that included cache

memories per core. Since the last few decades, shared parallel caches were taken into account in existing simulators, providing diverse types of simulation with timing and latency models.

At another simulation level, cycle-accurate simulators represent the simulation of a microarchitecture with the accuracy of a cycle execution and respect the execution time of an instruction. Thanks to this kind of simulator, it is possible to correctly evaluate the execution time of applications. They provide a timed simulation, making it an ideal solution for also testing new architectural design choices in an accurate way [RCBJ11]. The drawback is that they can be very slow as the simulation is detailed [WM08]. The event-driven simulator gem5 [BBB+11] proposes a timed cache model with a complete memory hierarchy with coherence protocols. It supports multi-core cache models but as the simulation of the virtual cores is single-threaded, the resulting global simulation is extremely slow.

When it comes to software centric systems, the timing is not necessarily as important as one believes. Instruction Set Simulators propose a simulation at a higher level of abstraction, at the granularity of an instruction. It results in a less accurate but faster simulation than with cycle-accurate simulators. Instruction Set Simulators are better used for the validation of new design choices of Instruction Set Architecture due to their speed performance. The produced simulation is, however, in essence functional.

2.1.3 Cross-ISA simulation of software centric systems

Instruction set simulation offers the possibility to do cross-ISA simulation and is the primary tool for validating ISA choices, retargeted compilers, Operating System ports and pre-silicon validation of hardware/software systems. With this kind of simulation, the word target refers to the Instruction Set Architecture (ISA) of the simulated architecture and the word host refers to the architecture of the machine on which the simulator is run. Imposing the target and the host to have the same architecture is too much of a constraint. In consequence, Instruction Set Simulators (ISS) are a satisfactory solution to test and evaluate software centric systems of multiple architectures on a single host. Instruction set simulation also had a significant role in guiding system-level design space exploration. Indeed, booting an Operating System on a platform and running significant applications requires a full-system simulator that is both fast and, at least functionally, faithful. Usually a full-system simulator, meaning a simulator able to simulate all the environment an Operating System needs, is built on an Instruction Set Simulator to achieve good performance and also because all the implementation details of the devices are not needed.

Instruction Interpretation is a well-known and simple technique but is used less through the years because of the relatively low performance it reaches. It interprets and executes one instruction at a time, which explains why the execution time can be really slow. The mechanism that consists of transforming instructions from the ISA of the target to instructions of the ISA of the host is referred to as Binary Translation. About 30 years ago, Binary Translation was an emerging technology. It was helpful in particular to run an old binary on a new architecture that was compiled on an ancient one. If the source code was missing for whatever reason, it was a suitable alternative to continue using the binary.

Binary Translation can be done statically. With this technique, all the instructions of the target are translated into host instructions before execution. Figure 2.3 illustrates

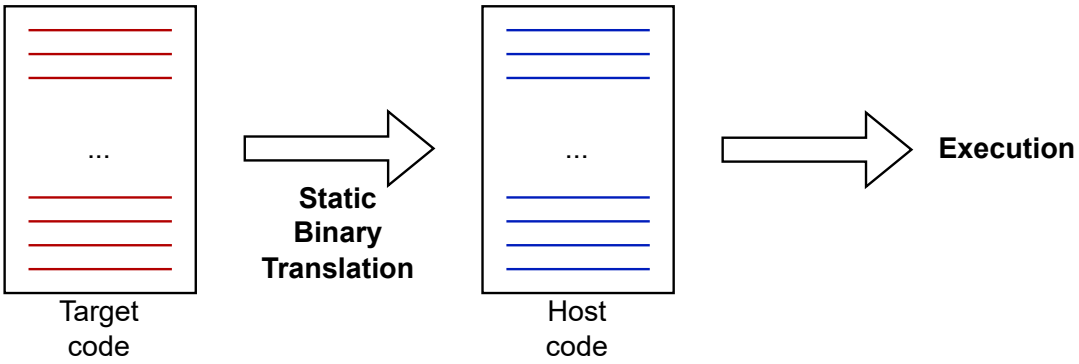


Figure 2.3: Simplified representation of the **Static Binary Translation** mechanism

the simple mechanism of Static Binary Translation. Even if it produces better performances than interpretation, Static Binary Translation does not deal with self-modifying code that happens for instance when loading dynamic libraries. Moreover, sometimes the value of data in the code can only been determined during run-time, for example because of indirect branches. The limits of Static Binary Translation were already known 3 decades ago [CM96].

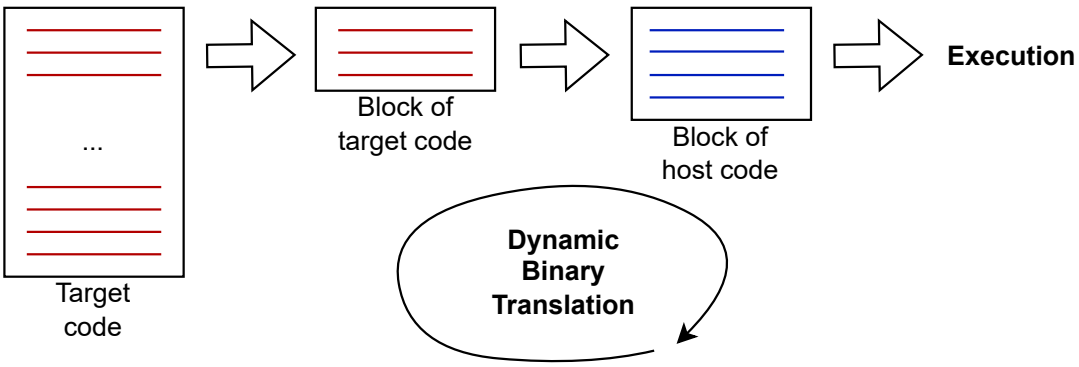


Figure 2.4: Simplified representation of the **Dynamic Binary Translation** mechanism

Dynamic Binary Translation appears to be the solution to deal with these problems. The principle is presented in Figure 2.4. With this mechanism, the target code is translated and executed by blocks of instructions. One of the major differences is that translated blocks are stored to be reused if the same block of instructions appears again and as a result it minimizes the overhead induced by the translation.

2.1.4 Simulation trend: from mono to multi-core systems

With the trend around multi and many-core systems, ISS research attempted to improve simulation by proposing parallel simulation approaches. However, even if ISS mono-core simulation was a mature technology already long ago, work needed to be done to follow the research and production of many-core systems [ABvK+11]. The first attempts at a multi-core simulation with ISS resulted with a sequential simulation of the virtual cores. Each virtual core is a process and the simulator executes them one by one following a given algorithm. Simulating a high number of cores in this configuration creates a bottleneck that greatly affects the performance and is no longer a viable solution given the emerging many-core architectures that need to be simulated. As

seen on Figure 2.1 that outlines the structure of a recent server composed of 24 physical cores, nowadays systems such as the small embedded ones and personal laptops are as well composed of multiple cores. Therefore, the best implementation one can think of is to take advantage of the parallel architecture of the host machine. This results in a more scalable simulation and less degradation of the simulation performance. Now that servers are produced with many-core processors (up to 128 or 256 cores), having this kind of parallel simulation is a requirement. A lot of DBT engines currently provide support for parallel execution, using the physical cores of the host machine to run simulation threads. The combination of the parallel execution and the sequential speed it provides make the DBT the best solution among the ISS methods to do full-system simulation. In the end, the DBT is the best solution for cross-ISA simulation of software centric systems since it offers a speed/accuracy trade-off that developers find particularly useful.

2.2 Dynamic Binary Translation

2.2.1 Mechanism overview

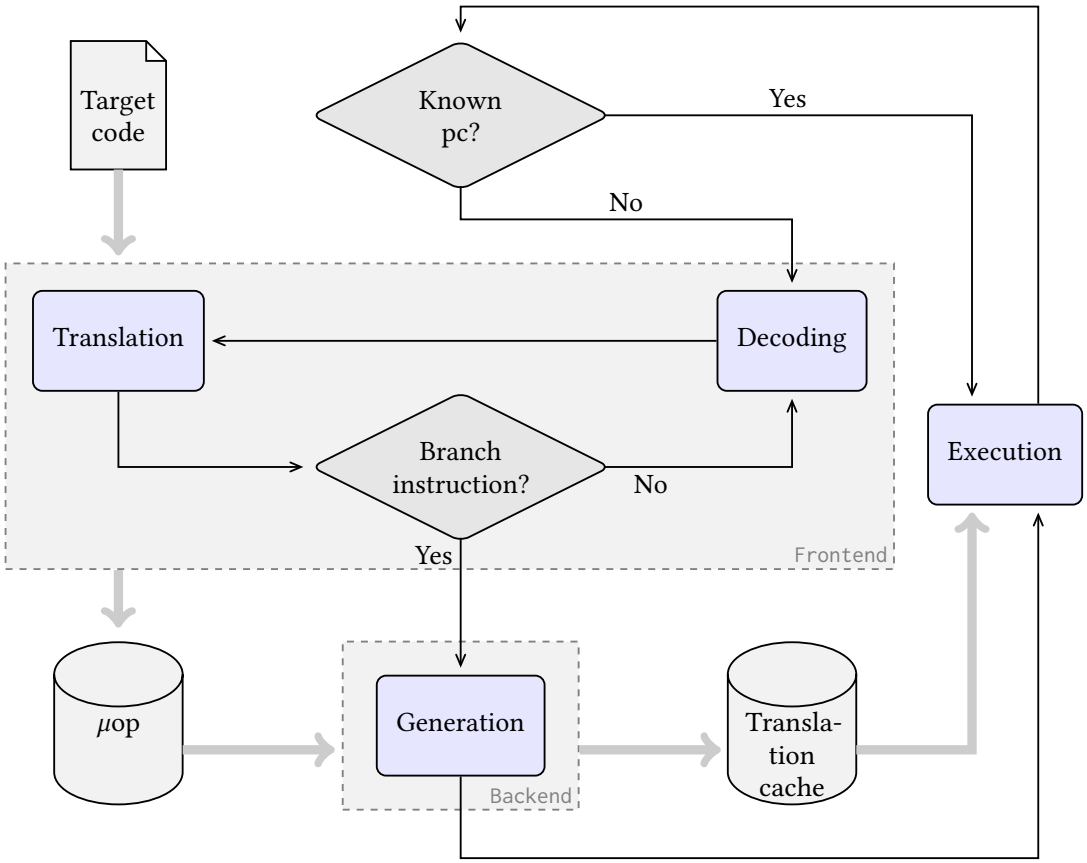


Figure 2.5: Overview of the DBT mechanism inspired from Luc Michel's thesis [Mic14]

Figure 2.5 illustrates the DBT mechanism. One by one, the target instructions are fetched and translated into micro-operations which are an intermediate representation that only the DBT understands and they are stored in a buffer (Frontend). We continue this translation until we have an instruction that is a branch, meaning it jumps to

another address. Once we have a branch instruction, a block called Translation Block (TB) is created. Then the TB is translated into host instructions and can be executed on the host machine (Backend). The translated TBs are stored in a translation cache so that it avoids doing all the translation steps again and it bypasses the overhead of translating a block again. If the address of the program counter has already been seen, the corresponding TB can be directly executed on the host. As the cache has a limited size, selected blocks will be evicted when the cache is full.

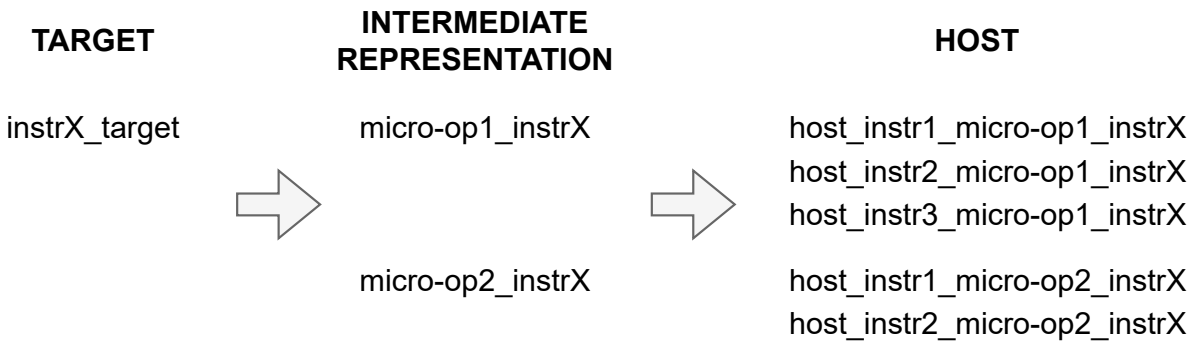


Figure 2.6: Target to **Host** code generation principle

```
1  IN: Target instructions                  OUT: Host instructions
2  0x00000000000325d6:  add   a5,a5,a3      -- guest addr 0x00000000000325d6
3  OP: Intermediate representation          0x7f5204000f93:  movq     0x68(%rbp), %r12
4  ...                                      0x7f5204000f97:  addq     %r12, %rbx
5  ---- 00000000000325d6                    0x7f5204000f9a:  movq     %rbx, 0x78(%rbp)
6  add_i64 x15/a5,x15/a5,x13/a3             ...
7  ...                                      ...
```

Figure 2.7: Translation process of a single **add** instruction

Figure 2.6 presents the generation from the target to the host of a generic instruction. It always takes more host instructions to describe target instructions. Each target instruction corresponds to multiple micro-operations and each micro-operation corresponds to multiple host instructions. In this example, one target instruction generates 5 host instructions but it varies a lot depending on the target instruction. Figure 2.7 shows a concrete example of the DBT principle of the QEMU DBT engine on a single `add` instruction for a RISC-V target and a x86 host. The `add` instruction in RISC-V is first translated into the intermediate representation of the DBT and finally translated into x86 instructions to be executed on the host. Even if we represent here only one target instruction, in practice this `add` instruction is inside a TB and the corresponding host instructions will effectively be executed when the entire TB executes. In this example, we easily transform an `add` instruction in RISC-V into an `add` instruction in x86 but some instructions are more complex to translate. Sometimes it is not possible to directly describe a target instruction with host instructions because they are too complicated. This is the case for memory accesses. To bypass this problem, the behavior of these instructions is described in C functions and these functions are called during the intermediate representation. Figure 2.8 represent the DBT principle of the QEMU DBT engine of a target load instruction in RISC-V for a x86 host. One can notice the call of a function `qemu_ld_i64` inside the intermediate representation.

```
IN: Target instructions                  OUT: Host instructions
0x0000000000032776: ld s9,8(a0)          0x7f52040010db: leaq     8(%rbx), %r13
OP: Intermediate representation          0x7f52040010df: movq     (%r13), %r13
...                                      0x7f52040010e3: movq     %r13, 0xc8(%rbp)
---- 0000000000032776                    ...
add_i64 tmp3,x10/a0,$0x8                 ...
qemu_ld_i64 x25/s9,tmp3,leq,0
...
```

Figure 2.8: Translation process of a single **load** instruction

Regarding parallel execution, the main idea is to create a thread for each core simulated. QEMU [Bel05] proposes a stable parallel implementation since 2016. The idea behind the implementation is to avoid doing a lot of locking so that the simulation time degradation is kept under control. The translation of the TBs is done in each thread that simulates a core and then the translated TBs are stored in a shared structure to be reused by other threads if the translation is already done. When a thread wants to retrieve a TB in the shared structure, atomic instructions are used. In addition, atomic operations are simulated using the host instructions. This is quite tricky, as the memory consistency models of the target and host might have different constraints, but the current implementation is stable. Moving from a mono-core to a multi-core simulation thus eased the simulation of many-core systems and QEMU was not the only DBT engine to propose a such execution.

2.2.2 Instrumentation

Instrumentation tools are useful tools to evaluate programs. They help to produce traces that show details of what happens during execution. Doing instrumentation in general can be tough, as it will induce a non-negligible overhead as pieces of code will be added. Basic Dynamic Instrumentation helps at retrieving information related to the instructions and memory accesses executed. Regarding full-system, it can be helpful to have information on the number of interrupts, page faults, context switches, statistics on the branch predictor and so on. As a result, instrumentation of full-system is now quite common and used. The Dynamic Binary Instrumentation framework Pin [LCM+05] provides an API to analyze dynamically programs during run-time and does not degrade the performance a lot. Multiple tools are now based on Pin such as VTune which helps examine the performances of software based on a Linux or Windows Operating System. However, Pin shows some limits on the devices and architecture it can instrument.

The integration of instrumentation inside simulators is an important asset to analyze and debug applications and new design choices. Functional simulation proposes an attractive simulation time as it has a good accuracy vs speed trade-off but adding instrumentation will induce an important overhead. That is why works try to do it in the least impactful way.

The smallest execution unit in the DBT is the instruction. The most straightforward approach to do instrumentation when using the DBT mechanism would be to insert a call to a function each time a target instruction is executed. Figure 2.9 illustrates this principle. This can be seen as identical to helper functions we presented just above. Information on the type of instruction (memory access or not, read or write, number of

```
0x7f1ffbe5692c: auipc a5,237568
                call_function()
0x7f1ffbe56930: ld    a5,-612(a5)
                call_function()
0x7f1ffbe56934: sb    s0,0(a5)
                call_function()
0x7f1ffbe56938: ld    ra,8(sp)
                call_function()
0x7f1ffbe5693a: auipc a5,270336
                call_function()
0x7f1ffbe5693e: sb    s0,1470(a5)
                call_function()
0x7f1ffbe56942: ld    s0,0(sp)
                call_function()
0x7f1ffbe56944: addi  sp,sp,16
                call_function()
0x7f1ffbe56946: ret
                call_function()
```

Figure 2.9: Instructions in a Translation Block of the DBT mechanism

bytes, addresses) can be passed through the function call. Nevertheless, calling a function each time a target instruction is executed will generate a non-negligible simulation time overhead. Indeed, a function call is more expensive than usual as it is necessary to save and restore certain registers that the DBT uses. Moreover, instrumenting the parallel implementation of the DBT is even more challenging as TBs are shared between the threads that represent the virtual cores. Therefore, instrumentation operations need to be done carefully if it concerns shared values (for example when counting the total number of executed instructions).

2.2.3 New hardware structures and the accuracy vs speed trade-off

By instrumenting Dynamic Binary Translation simulators, it becomes possible to add new architectural structures inside the simulation. The classical DBT mechanism allows one application compiled for a specific architecture to be executed on a host machine with a different architecture. The DBT simulator QEMU offers the possibility to simulate in addition some hardware components such as USB controllers and network devices that produce a full-system simulation. Adding new hardware structures inside the simulation improves the accuracy in a sense where the simulated system is more detailed and complete but on the other hand degrades the global performance.

Regarding the research on DBT instrumentation, a choice needs to be made on where to fix the limit between increasing the accuracy of the simulation by adding hardware components and keeping a fast simulation time. The idea is not to degrade considerably the performance achieved by the DBT mechanism but to model the component in a smart way to limit the loss of performance while having useful evaluations.

Let us take again the cache component as an example. This small amount of memory of a few kilobytes can be organized following three different strategies. The first one is called `Direct Mapped`. It can be seen as a vector and each block of memory will have a designated placement in it. The second one called `Fully associative` specifies that each memory block can be placed in any line of the cache. The third and last one is called `Set associative` which is a combination of the two strategies above. An

example is presented Figure 2.10. In this strategy, a memory block can be placed on a subset of lines in the cache.

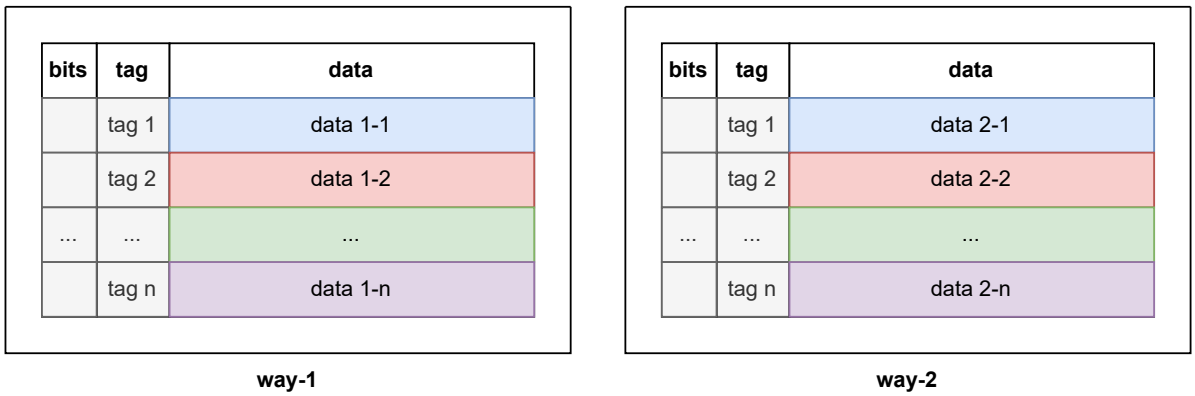


Figure 2.10: Simplified representation of a set associative cache with 2 ways

Figure 2.10 is a simplified representation of a 2-way set associative cache. The colors represent the subset of lines in which a memory block can be put in. Implementing a model for it in simulation can be done in many ways. To implement a detailed simulation of the behavior of such a cache, the solution is to model everything that is contained in the cache meaning the status bits, the tag and the raw data. This is mandatory to simulate the correct behavior. The most detailed simulation would also model the latency of the accesses done to the cache since finding the data that we are looking for in the cache (cache hit) will not take the same amount of time as if the data is not in the cache and needs to be stored in it (cache miss). However, to only produce statistics such as the number of hit/miss, the number of accesses or even the number of evictions in the cache, doing a fully detailed simulation is not mandatory.

2.3 Conclusion

We presented in this chapter why, in the scope of software centric systems, the DBT approach is in our opinion the best option for cross ISA simulation. We saw that multi-core systems are widespread in our daily life and are challenging to simulate at scale. Improving the timing accuracy of the simulation performed using the DBT mechanism by adding models of hardware structures is a tricky challenge, as these additions should not slow down drastically the high simulation speed it provides. By considering these factors, we can ask ourselves the following questions:

- **How can we assert that DBT parallel implementation scales well on the multi-core host machine?**
- **Can we rely on the host configuration to improve the DBT parallel simulation speed?**
- **Can we benefit from the DBT approach to design in particular a cache model that limits the impact on the global simulation time?**
- **How can we enhance the time accuracy of the DBT mechanism when adding models of new architectural features in the simulation without overly degrading simulation speed?**

Chapter 3

State of the Art

In this chapter we present the state-of-the-art regarding how to extend and accelerate simulation based on the Dynamic Binary Translation mechanism. As an introduction, we present some of the existing DBT engines. To be up to date with the emerging multi-core processors, DBT engines had to support parallel simulation. That is why we do an overview of the related work concerning parallel implementation of the DBT and we also focus on works that tried to improve the DBT mechanism by making it faster or more precise. Finally, as DBT-based simulators produce a functional simulation, we focus on the related works that add models of new hardware structure in DBT-based simulators to make them more accurate from a non-functional point of view, but with the drawback of degrading the global performance.

Table of contents

3.1 Dynamic Binary Translation (DBT)

Due to the sustained interest in systems simulation and the growing interest in cross-ISA simulation, DBT approaches continue to evolve. The fact that blocks of instructions are translated and reused during run time makes the DBT a satisfactory solution to simulate in a fast time a diversity of Instruction Set Architecture (ISA). This section first provides an overview of some DBT engines, then presents work done to support parallel execution of the DBT, and finally how the DBT mechanism has been extended and improved over the last few years.

3.1.1 DBT engines

At its core, the DBT mechanism aims at generating instructions to be executed on a host machine from instructions originating from a, generally different, target architecture. However, more than 20 years ago, only a few target/host associations were possible and there was a need to support other combinations. To circumvent this issue, [UC00] implemented a DBT-based framework to deal with multiple target and host machines, which translators of that time were not designed to do. This framework supports a wide diversity of Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC) architectures. It offers the possibility to give information on the specifications of the host/target machines to run the simulation, and new machines can be added easily (according to the authors) without many constraints. But in the end, the performance of the results was limited and lots of work needed to be done to support multiple target and host architectures in an efficient way. Currently, most of the DBT engines support several different target/host combinations.

User-mode vs System-level simulation

The DBT engines can be separated into two categories: user-mode level simulation and system-level simulation. The former one refers to simulators able to perform the simulation of an application compiled for a target ISA onto another host ISA. In this situation, the host Operating System takes care of the system calls using what is now referred to as semi-hosting, *i.e.*, by calling a wrapper in which marshalling of the call target parameters into host ones is done prior to calling the host system call. The latter one refers to the capacity of a simulator to run any software stack, including Operating System. Contributions have been made in the two categories as they are both useful depending on what needs to be checked or evaluated during simulation [WHK+07, OYTN17]. However, the contributions focus more on improving the system-level simulation because of the possibilities it offers. Indeed, as presented in [SWF20], system-level simulation, and more precisely the DBT one, is useful for the development of mobile software, for simulating old games as the game consoles do not always remain available, etc. Regarding system-level simulation, the machine simulator Embra [WR96] proposed in 1996 a system-level simulation based on the DBT mechanism able to model a MIPS processor so that an Operating System can run on top of it. More than 20 years later, system-level simulation is still an ongoing research topic. Indeed, in the current decade, the system-level DBT hypervisor named Captive [SWF20] appeared as a solution to support easily new target ISAs with particularly good performance. It is retargetable and shows better performance than QEMU by outperforming it by a factor of 2.

On the other hand, QEMU might be the most known and used in research. It has a large and continually active community and has been open-sourced from the start, which makes it an attractive engine to work on for researchers. It supports a wide diversity of target and host architectures, and is used in industry too, which is why it has become the reference of DBT engines. Its system-level simulation supports multiple hypervisors such as KVM (that is also developed and maintained by the QEMU community) and Xen or even the NetBSD Virtual Machine Monitor. QEMU is actively maintained by the people from the largest software companies working at the hardware/software interface, a guarantee of credibility and longevity.

3.1.2 Parallel implementation

The increasing number of cores integrated on System on Chips (SoCs) induced an important concern regarding simulation. To be in line with the improving multi-core systems, simulators needed to change the way they are designed to be able to reach good simulation speed to provide the required environment to do parallel applications analysis. Simulators needed to take advantage of the parallel architecture of the host if they wanted to achieve reasonable simulation performances. Determinism, and to a lesser extent functionally correct interlacing of threads executions, is hard to produce for multi-thread workloads. It is thus necessary for simulators to use synchronizations. However, having a lot of synchronizations will force a limitation on the number of cores simulated as it degrades the simulation performance. Native simulation was the center of interest of some research to produce a simulation that relies on the parallel architecture of the host [SHP12, DGVS14, NS15]. These works were driven by the fact that native simulation allows a simulation performance really close from the native execution performance. To reduce the impact of synchronization when multiple threads are running in the simulation, [DGVS14] propose a native multi-core simulation with asynchronous synchronizations. The synchronizations are done only when a shared element needs to be read. As a result, the simulation time is less degraded than when doing synchronization naïvely. [NS15] also uses the idea of asynchronous synchronizations to mitigate the overheads of parallel execution. But in the end native simulation has many flaws: it is overly complex to solve the guest/host address translation issue, it deals neither with guest assembly routines nor with self-modifying code that is fairly present in today's workloads. Overall, the DBT is a better option.

However, relying on the host for multi-core execution creates contemporary issues. When it comes to memory, load and store instructions cannot be directly translated from target to host instructions as they are too complex. Indeed, they must undergo address translation, which might mean at some point performing a simulated guest page walk. When the memory is shared, synchronization instructions are used and guarantee that memory accesses are done atomically. Part of code will be generated additionally in the simulation through functions that describe the behavior of these instructions. [KSC+20] addresses this problem by proposing a way to deal with load-link/store-conditional instruction simulation in the DBT process. Indeed, these instructions are used to perform synchronizations between multiple threads. Matching these simulated instructions with ones of the host architecture is challenging and if not done properly, will create execution errors according to the authors.

The scalability of such parallel implementations also needs to be considered when targeting many-core systems. [ABvK+11] presents a parallel DBT implementation tar-

geting the ARCompact ISA. They can run the simulation up to 2048 virtual cores and their main idea is to create one thread per virtual core. A cache is also created to share translation elements. By doing so, they result with a very time-attractive parallel simulation, although very ad-hoc.

About ten years ago, [WLC+11] and [DCHC11] started working on the parallelization of QEMU implementation. By simulating the atomic instructions of the target by using wisely the host atomic instructions, they limited the overhead of doing synchronization between the threads that simulate the target virtual cores. However, this started raising novel issues. A slight difference of semantic within the atomic instructions between the target and the host will induce lots of code to be generated as the translation of these instructions will need to pass through helper function to mimic the target behavior as precisely as possible. Doing parallel execution with one thread per virtual core also raises the question on how to schedule the threads execution. Since QEMU has a very living community, the code source grows regularly. As a result, modifications and improvements of QEMU need to be done carefully to ensure the quality and maintainability of the source code. Consequently and unfortunately, these works were not upstreamed in QEMU. Few years later, [DBK+16, RSR16, CBBC17] reworked on the idea of a parallel implementation of QEMU by proposing a more viable version and finally a solution was accepted to be upstreamed in QEMU [Ben20] which was named Multi-Thread Tiny Code Generator (MTTCG). The first target was originally the ARM ISA and the host a x86-64 architecture. The principle of the MTTCG implementation illustrated Figure 3.1 (and inspired from [Ben15]) was pretty basic. Each thread of the virtual CPU executes on a different host CPU. The main goal of this principle was to minimize the locking. To do so, the translation of the blocks is done locally first and then are put in a shared cache to be reused later by other virtual cores that need the translation. When the cache is full, all the stored translation blocks are flushed and translations need to be done again. Furthermore, most shared structures are updated by using atomic instructions.

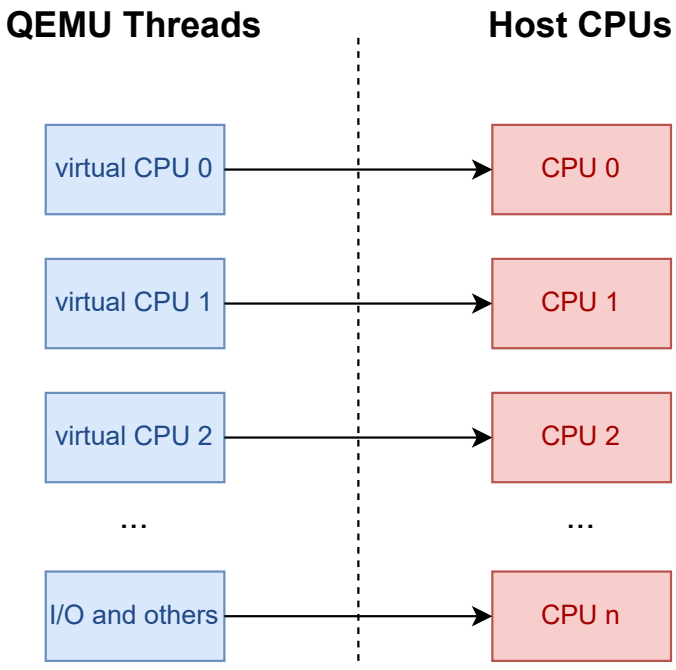


Figure 3.1: Simplified representation of the Multi-Threaded Tiny Code Generator principle

3.1.3 Improving DBT mechanism

To follow the evolution of processor centric systems designs, improving the DBT mechanism continues to be an important concern. The goals when improving the simulation are on one hand to make it even faster and on the other hand to add new non-functional models to make the simulation more accurate when used for architecture evaluation. In his thesis, Antoine Faravelon [Far18] outlined the techniques to accelerate the DBT by presenting how people optimized the code generated, some multi-threaded implementations and memory access simulation accelerations. His thesis subject itself takes part in the state-of-the-art on how to accelerate the DBT mechanism as he proposed techniques to accelerate the memory accesses [FGP21]. We will give here an overview of some DBT improvements, but without going into so much detail as in Faravelon's thesis.

Accelerating the DBT principle

One way to improve the DBT mechanism is to optimize the generated code. To do so it first requires annotating the code. Then, thanks to that, modifications can be made during the simulation runtime. By relying on the existing Dynamic Binary Instrumentation framework DynamoRIO [Gar14], [HDBZ15] implemented DBT optimization techniques for dynamically generated code of target applications. They showed better performance than the state-of-the-art of dynamic binary translators of that time for just-in-time applications. But we will not describe this topic more as the next section will give more details on the existing Dynamic Binary Instrumentation frameworks.

Another way of improvement is to rely on the host hardware to accelerate the translation mechanism. [RRD17] worked on this topic and presented a DBT approach in which parts of the mechanism are accelerated thanks to hardware components. They spotlight the fact that the scheduling is the part that will take the most time in the DBT process. That is why they used a VLIW-based hardware accelerator to schedule the instructions generated during the Intermediate Representation stage as their host is a VLIW processor. In total they relied on three hardware accelerators to make the DBT mechanism faster. Thanks to their idea, the results showed non-negligible speed-up compared to the classical version and less energy consumption. However, designing such a system can be very complex, the improvements benefit only a subset of the possible ISAs, and important matters such as address translation, supervisor code simulation, and so on, still need to be dealt with.

Implementing a multi-threaded DBT engine can be seen as a way to accelerate the process [HHY+12]. Dividing the TB execution into multiple threads will for sure boost the execution time and will produce speed ups. However, we dedicated an entire Section 3.1.2 on parallel implementation so we will not discuss this topic further here.

Last but not least, one of the main bottlenecks of the DBT process are the indirect branches. For direct branches, the address to jump at is known during the translation stage. But for indirect branches, the address is determined only at runtime. [dGGL16] worked on MAMBO-X64, a DBT-based engine to optimize the translation of indirect branches thanks to three efficient techniques. Each technique concerns an indirect branch source, and the sources cited in the article are branch tables, function returns and function pointers. Even if indirect branches do not represent a large part of the execution, their techniques help at reducing the overhead of the translation by a non-negligible percentage.

Improving the accuracy of dynamic binary translators

Another path to improve the DBT mechanism is to increase the accuracy of the simulation when it is used for the performance evaluation of the target system. This process will for sure degrade the execution time but the simulation will be more precise so it can be a good trade-off depending on the usage. Improving the accuracy of DBT engines can be done in lots of diverse ways. Adding the simulation of new components in DBT engines is a way of doing it. Classical DBT engines do not include the simulation of all the microarchitectural details of a processor since the philosophy of the DBT mechanism is to produce a purely functional simulation. However, it can be interesting to do so to keep the fast execution time that the DBT provides and avoiding using other simulators that can be terribly slow to execute. In this thesis, our interest is focused on creating a cache simulator inside a DBT engine and we give Section 3.3 a detailed presentation on the existing cache simulators and we focus on the ones related to DBT engines.

Because DBT engines do not give microarchitectural details, [BFT10] proposed in their work to associate a cycle accurate processor targeting a ARCompact ISA with a DBT simulation. Thanks to that, they provide a complete microarchitectural environment. Their idea is to update the microarchitectural details of the processor each time an instruction executes. The resulting model thus executes instruction by instruction (philosophy of ISSs mechanism) while being accurate in term of microarchitectural details and shows better performances than simulation based on FPGA. However, their work is limited to mono-core simulation.

In their work, [CNZ20] spotlight the issue on full-system simulators where some applications need to be accurately timed. To keep a fast simulation time, they created mcQEMU, a time-accurate simulator based on QEMU in which they added time models. QEMU already has a partially timed simulation mode named ICount. As the name implies, this feature counts the number of instructions executed. They used this feature in QEMU to implement a more complete timed model. However, the ICount mode does not deal with multi-threaded execution. To make all the threads consistent in time, each thread has its own virtual time and an algorithm is applied on top of that to perform synchronization at every ”event” (that follows the principles of Parallel Discrete Event Simulator). To validate their work, they targeted an ARM processor and showed an error of only 15% of time accuracy when comparing to the NXP i.MX6Quad processor (ARM processor) with the TACLeBench benchmarks suite.

To conclude, improving the DBT mechanism is a very diversified topic. As it provides a functional simulation, having a DBT translator which is extremely fast is attractive for researchers and for the software industry. Adding models of hardware structures in the process will definitely induce an overhead on the simulation time. But in the end, we have a good trade-off as we can still take benefits of the execution speed that the DBT provides and the simulation accuracy will be better.

3.2 Dynamic Binary Instrumentation (DBI)

Instrumentation is a crucial tool to analyze software to retrieve any kind of information such as memory accesses and thus improving them regarding the topic of interest. Instrumenting applications in general can be very particularly challenging to do as it will for sure (a) degrade the performances of the targeted application and (b) disturb

the code flow and change the state of the hardware structures, as pieces of code need to be added. Reaching good performances is one of the main goals. Compared to Static Binary Instrumentation, Dynamic Binary Instrumentation allows to do instrumentation and execution of an application at the same time, which makes it the approach of choice today.

About 30 years ago, the Instruction Set Simulator Shade [CK94] introduced a way to do fast profiling. They highlighted the goals that good instrumentation tools must achieve: ease of use, diversity of software it should instrument, fast execution time, availability of lots of details and finally support to instrument architecture not yet created. Shade can dynamically cross-compile and execute applications of a target architecture directly on the host and also cache the translation code which recalls the principles of Dynamic Binary Translation. A tracing tool is integrated with the Instruction Set Simulator and the user can specify what information he wants to retrieve.

In the scope of the Dynamic Binary Translation, doing instrumentation efficiently is the concern of various works. To meet the demands on an efficient and portable instrumentation tool, [LCM+05] proposed 20 years ago Pin, a convenient instrumentation framework. Similar to ATOM [SE04], Pin has a diversified API and is also able to support a large diversity of architectures. The user can easily decide where to add call to instrumentation functions in the application. The results show that Pin can be faster for some configurations than other well-known dynamic instrumentation tools of that time: Valgrind and DynamoRIO. However, a few years later, Valgrind [NS07] announced to be a framework better than Pin and DynamoRIO for heavyweight Dynamic Binary Instrumentation.

According to [LHW+14], most of the state-of-the-art Dynamic Binary Instrumentation tools only support the same ISA for the target and the host. Having the target and the host machine with the same ISA can be too restrictive of a constraint. Because the vast majority of the existing computers are based on a x86 architecture, it became interesting and useful to be able to instrument for example RISC-V or ARM applications on a x86 machine without the obligation to have a RISC-V or ARM machine. Thus [LHW+14] presents DBILL, a cross-ISA Dynamic Binary Instrumentation framework that uses LLVM as a backend. DBILL also relies on QEMU by leveraging on HQEMU [HHY+12], a multi-threaded Dynamic Binary Translator implemented by the same authors. Since DBILL is presented as a heavyweight DBI tool, they did experiments to compare DBILL with Valgrind. The results show that DBILL is more than 8 times faster than Valgrind with ARM benchmarks. [CC19] also decided to focus on this topic. In this paper, the authors present two techniques to improve the performance of cross-ISA simulation and one technique to do instrumentation efficiently. They implemented their techniques in Qelt, a DBT-based simulator based on QEMU. When doing complex instrumentation, they showed that they can reach the performance of the DBI tool Pin and they are faster than most of the state-of-the-art cross-ISA DBI tools when doing full-system instrumentation. To finish on the cross-ISA DBI topic, QEMU also proposed a framework to do instrumentation [Gui11]. It has now been given the name of Tiny Code Generator (TCG) Plugins and was developed by the authors of [CC19].

As QEMU is without doubt the most used DBT-engine and thus is an interesting choice for us, we will detail here the functional principles of the TCG Plugin, a QEMU feature that helps to do instrumentation at various levels.

QEMU TCG Plugin

The Plugins provide the user an API to do code instrumentation relatively easily and efficiently. Information can be retrieved at the different stages of the simulation process, *i.e.*, during translation or at execution time. Because QEMU is based on the Dynamic Binary Translation principle, the smallest unit that it is possible to instrument during simulation is the instruction. Through the API, the user can subscribe to various kinds of events. The most classical events are: a Translation Block (TB) translation, a TB execution, an instruction execution and the execution of a memory access. In addition, it offers the possibility to subscribe to events such as a virtual core initialization, a virtual core exit, when a virtual core goes idle or when a virtual core resumes. Regarding instruction execution, it is also possible to instrument only particular instructions. Calls to plugin functions are inserted during translation for all possible events. Thanks to that, it avoids doing the translation a second time to add the calls to the plugin functions. The calls to events that are finally not used are removed eventually by a later optimization pass.

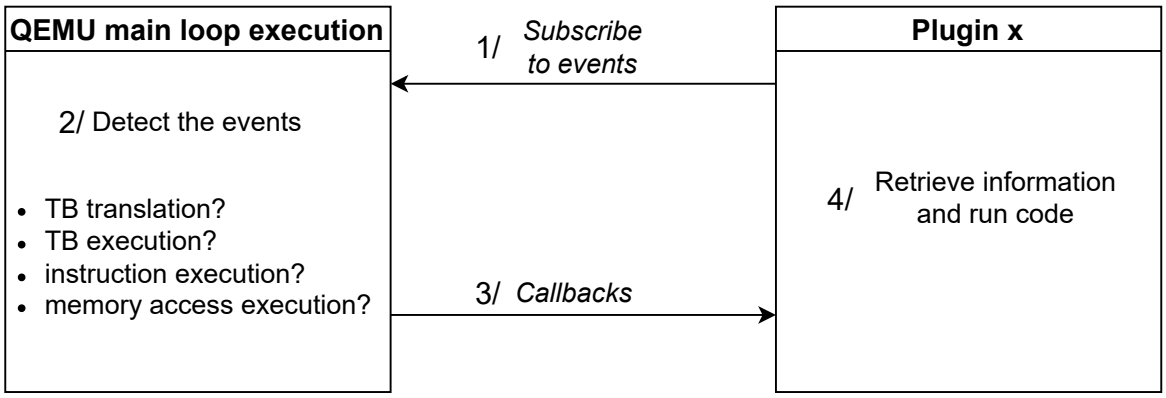


Figure 3.2: Simplified representation of the QEMU TCG plugins mechanism

Figure 3.2 illustrates the TCG plugins mechanism simplified. The first step for the user is to create a new C file. Then the user needs to choose to which events he wants to subscribe to, using the functions of the plugin API. During the execution of the main loop of QEMU, each time the event to which the user subscribed is detected, information is sent back to the plugin through callback functions. We represent on Figure 3.2 only the classical events. Finally, once the user has retrieved the information corresponding to an event, it can add code in the plugin file to compute what he wants to compute. Currently a dozen plugins are upstreamed in QEMU.

One of the main advantages of the TCG plugins is that they are architecture independent, meaning they work with all the different targets supported by QEMU. Moreover, the instrumentation works also with the MTTCG (Multi-Threaded TCG). When the plugin code is called, it is done inside of a lock that protects QEMU's internal structures only. Therefore, if some data in the plugins is meant to be shared between the virtual cores, for example a counter for the total number of instructions, the user needs to use his own locking scheme.

To summarize, the QEMU TCG Plugin API offers a relatively simple and efficient way to instrument any applications in the case of DBT-based simulation. The user can create any kind of plugin he wants based on the available events. As an example, in their paper [MCDK21] presents a methodology based on QEMU to explore In-Memory Computing architectures at the ISA level. They used the QEMU TCG Plugins to model some components thanks to the events proposed by the API. However, there is currently

limited access to processor specific information, such as the registers values, but a processor agnostic solution has been found to solve this issue [Oda23].

3.3 Adding new models: overview of cache simulators

Simulating models of new architectural features can be decoupled from the main simulator. As an example, in their work, [JMW+22] proposed to decouple the processor simulation and the simulation of the memory. By doing so, they were able to keep the accuracy of the simulation while providing a better simulation speed. Even if this technique shows satisfactory results, it will not be discussed more in this thesis as our focus is on adding architectural features directly in DBT-based simulation. But we will make an exception for cache simulations as it is our main interest.

Instrumenting applications through DBT-based simulators helps with retrieving information that can be used to simulate the behavior of hardware structures. For example, the QEMU TCG Plugins provide a plugin that simulates a cache model. By having all the instructions executed by an application, it is then possible to simulate the behavior of a cache. A cache is now a piece of hardware that exists in all present-day processors as it drastically improves the memory accesses speed. A survey on a quite large list of cache simulators was made by [BKP20] in 2020. As explained in this survey, about 30 years ago cache simulators were mainly used for research purposes and were the alternative for when the corresponding hardware was not available. Now with the increase of the use of simulation in industries since the hardware is not always easy to have access to, cache simulation is very much in use. Our interest is thus focused on improving the DBT by adding a cache model. That is why we do in this section a non-exhaustive overview of existing and well-known cache simulators and what has been done lately regarding DBT-based simulators. We divided the rest of this section into two parts. We will first do a quick overview of classical standalone, trace based, cache simulators and then we will see in more detail the existing cache models used inside simulators and DBT-based simulators.

3.3.1 Standalone cache simulators

The most classical cache simulators only take as input the trace of the memory accesses of an application. It is the case of Dinero, which might be the oldest and most known mono-core cache simulator that is still mainly used for educational purposes. Dinero IV [EH98] is the latest known version. There are now a multitude of cache simulators whose functionalities are similar to Dinero. We can for example cite Pycachesim [Ham15] which we also used to validate our results in mono-core. Last year, a work was published [VMK22] that integrates Pycachesim with QEMU to simulate in mono-core the instruction cache and the authors are planning to extend their implementation to a data cache simulation as well. Thanks to the module named Cachegrind [Wei08], the DBI tool Valgrind can also simulate classical cache models. These kinds of simulators are now useful for the research on enhancing the software stacks. Even if these tools are easy-to-use and provide a straightforward way to analyze the behavior of applications, they have their limits. Simulating a cache model inside the simulation of a complete architecture can be more interesting to understand and investigate how the cache interacts with the other components.

3.3.2 Cache simulation inside simulators

Cache models can vary a lot depending on how they are designed and what details they provide. [Mag97] proposed about 20 years ago a simulation of an instruction cache by extending the Instruction Set Simulator SimICS. Predictably, adding the instruction cache simulation degraded the performance of SimICS but the authors claim that the accuracy gained with the cache model justifies the overhead created in the simulation. More recently, some full-system simulators also implemented their own cache simulator, included with the simulation of other components. Some of them are called timing simulators and can reproduce the latency of the movements of data in the different cache levels. Gem5 [BBB+11] is one of the most popular microarchitecture multi-core simulators and provides a timing cache simulator. However, these are the ones that will induce a non-negligible overhead. By relying on an approximate timed hardware/software co-simulation that produces a good balanced functional and timed simulation, [CPVM10] implemented an instruction cache model that does not damage the simulation speed. Indeed, in the case of native simulation (simulation used by the authors), modeling a cache based on a tag search will damage the simulation speed too much. To stay in the other of speed of native simulation, the authors combined a technique to instrument the source code and a specific instruction cache model.

As the DBT mechanism supplies a functional but fast simulation time, it was interesting for researchers to try to implement a cache model at a high level of abstraction inside DBT-based simulators. [VDTT14] proposed a module inside QEMU to do a complete cache simulation and claim to be faster than Valgrind. However, they were not the first to work on how to include a cache simulation in QEMU [GFP09]. The drawback is that these works were very intrusive in the QEMU code and architecture dependent. Thanks to the non-intrusive implementation of the QEMU TCG Plugins and the easy-to-use advantage, a cache plugin was upstreamed during summer 2021 [Man21] and is currently able to simulate a simple cache model per virtual CPU. We now detail how this cache plugin is implemented.

QEMU Cache Plugin

Figure 3.3 presents the simplified mechanism of the existing QEMU TCG plugin that models the behavior of a cache. The plugin simulates for each virtual core a L1 instruction, a L1 data and a unified L2. The writing policy supported is Write Back Allocate and three eviction policies are supported: FIFO (First In First Out), LRU (Least Recently Used) and Random. The plugin subscribes to the events ”instruction execution” and ”memory access execution”. Each time an instruction or a memory access is executed, the cache plugin code is run and checks if it is a miss or a hit. Statistics variables are thus updated for each instruction execution and displayed at the end of the simulation.

The way the plugin is implemented is the naive way of doing it. Thanks to the simple API provided by QEMU, it is easy to run the cache simulation each time an instruction or a memory access executes. However, it also implies that a piece of code will run additionally each time a target instruction is executed in the simulation and it results in a non-negligible execution time overhead compared to the vanilla execution.

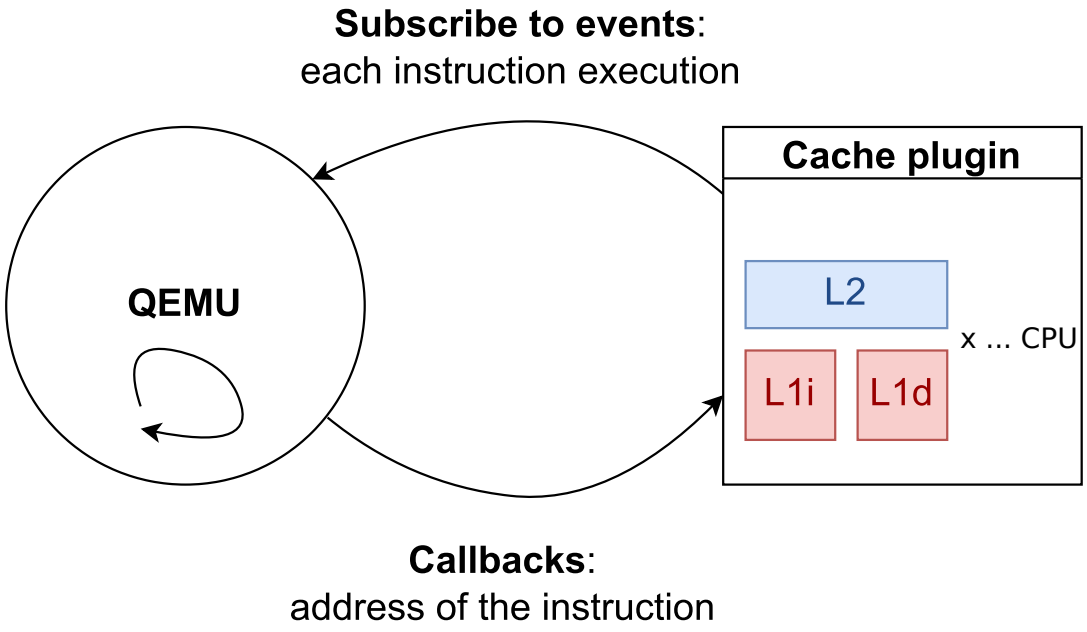


Figure 3.3: Simplified representation of the QEMU **cache** TCG plugin mechanism

3.4 Conclusion

Improving the DBT mechanism can be done in lots of diverse ways. A subpart of this field focuses on making DBT-based simulators even more faster. The parallel implementation in itself of these simulators is an improvement and makes the simulation run faster when relying on the host parallel architecture. We focus in this thesis on how to enhance the parallel simulation time. Chapter 4 presents our contribution for the scalability of QEMU parallel implementation.

When it comes to improving the accuracy of the simulation when specific metrics are to be evaluated on the target, many paths can be taken. Adding the simulation of hardware structures in a functional simulation requires to first instrument the code efficiently to have the needed information to simulate the behavior of a new component, which fortunately is possible in QEMU. In this thesis, our focus is on the simulation of a cache at a high level of abstraction. Chapter 5 presents our contribution on a cache simulation in QEMU with the TCG Plugins.

Chapter 4

To Pin or Not to Pin: Asserting the Scalability of QEMU Parallel Implementation

THIS chapter gives an answer to the questions "**How can we assert that DBT parallel implementation scales well on the multi-core host machine?**" and "**Can we rely on the host configuration to improve the DBT parallel simulation speed?**". We unsurprisingly chose QEMU [Bel05] as the DBT-based engine. We discuss here the scalability of the parallel implementation of QEMU which was never addressed to the best of our knowledge. Our contribution is this chapter relies on the experimental study of the scalability of the parallel implementation of QEMU with the possibility to pin a virtual CPU of the target to a host CPU. The idea behind that is that it will reduce the global simulation time. This work was done in collaboration with Saverio Miroddi who at that time already had on github an implementation of the pinning [Mir21]. We thus relied on his implementation to do the study.

In this chapter, before thinking on how to elaborate a thread affinity in QEMU, we first look at how the parallelism is done in QEMU. Then we see how to implement the pinning in QEMU and how to decide on which physical CPUs we do the pinning. Lastly, we present the possible impacts on the scalability regarding the host machine and the benchmarks and what simulation strategy we opt for.

Table of contents

4.1 QEMU Parallel Implementation

The Tiny Code Generator (TCG) is the tool inside QEMU that does cross compilation in order to run a target architecture that might be different from the host architecture. The generator is separated into 2 parts: the first one will generate the Intermediate Representation from the target code, which is a sort of assembly code that only QEMU understands and the second one will generate the final host code from the Intermediate Representation. Prior to 2015, the implementation did not take benefits from the parallel architecture of the host. When the simulation was run with multiple virtual CPUs, the execution was done on only one thread with a round-robin scheduling. This led to a clear scalability issue, given the large number of computing platforms that were embedding several if not many processor cores already at that time. Furthermore, at that time, I/Os peripherals and other tasks of the system apart from the CPUs were already handled in a separate thread. Figure 4.1 inspired from [Ben15] shows a simplified version of this mechanism where one can see that the virtual CPUs execute sequentially on one thread and a second thread is dedicated to the other tasks of the system. Here we arbitrary represent on the Figure the execution of the thread of the virtual CPUs on the CPU 0 and the thread for the other tasks on the CPU n but in practice it can be on the other host CPUs. The scheduling algorithm of the Operating System of the host decides on which CPU the threads execute. The algorithm is also free to change of host CPU multiple times during the execution of QEMU if it considers that it will improve the performance.

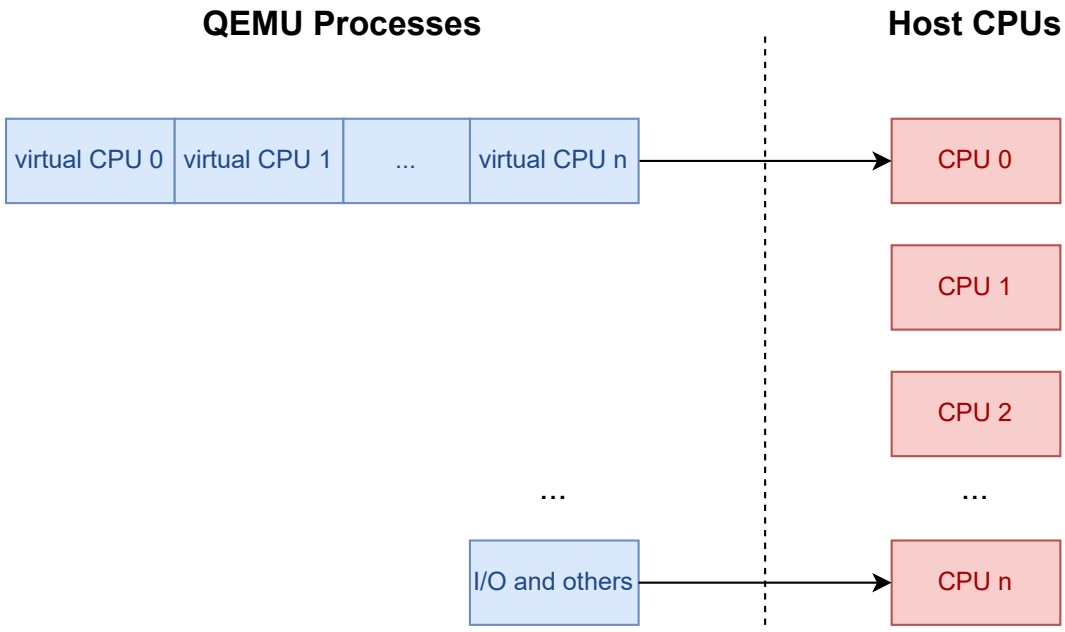


Figure 4.1: Simplified representation of the Tiny Code Generator principle (round-robin) with the peripherals thread

After that, the Multi-Threaded TCG (MTTCG) was designed and implemented and each virtual CPU now executes on a separate thread. Figure 4.2 presented in the previous chapter illustrates the process. As for the previous Figure 4.1, we arbitrary chose the assignment of the virtual CPUs on the host CPUs. In practice, the scheduling algorithm of the Operating System will do the assignment and it can also vary during the

execution of QEMU. However, the QEMU API allows the user to return to the previous single-threaded model with some command line options.

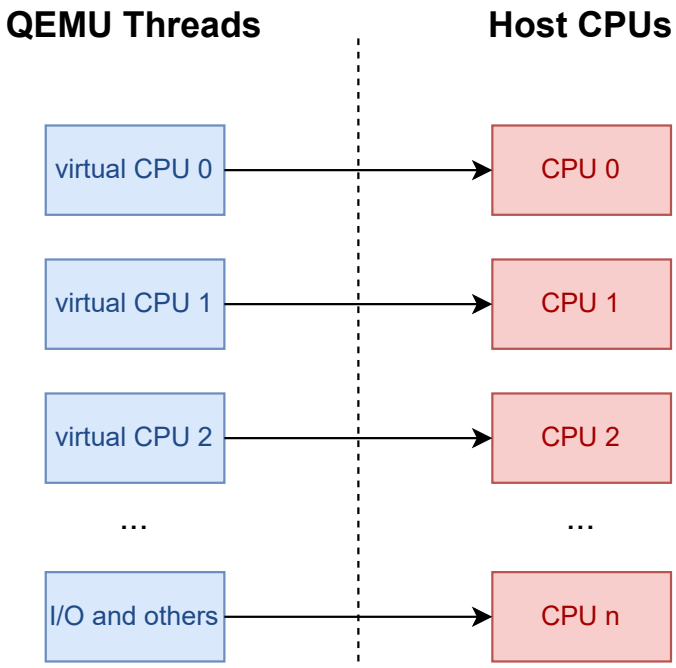


Figure 4.2: Simplified representation of the Multi-Threaded Tiny Code Generator principle

Migrating from the single-thread to the multi-thread solution in QEMU affected how the Translation Blocks are managed. Now, with the MTTCG principle, the translation of the blocks happens in each virtual CPUs. But in order to reuse information and to not do useless work, the virtual CPUs need to gather in once the blocks that they have already translated. Thanks to that, the other virtual CPUs can have access to blocks already translated by other virtual CPUs and thus avoid losing time of doing the translation. Compared to the single-thread TCG principle, the MTTCG principle added shared data structures between the threads. In fact, a single code generation buffer holds already translated blocks (TBs). When a virtual core executes, it will retrieve the TB corresponding to the target code in the shared buffer if it exists. If it does not exist, the translation is done locally in the thread to avoid using locks and the resulting TB is put in the shared buffer. The shared buffer is entirely flushed if full and the virtual CPUs need to start the translation again. Figure 4.3 illustrates the buffer that contains already translated block shared between all the virtual CPUs. Atomic operations are also used to reduce the impact of protecting the shared structures.

The transition to the multi-threaded solution however made the use of the instruction counting mechanism inconsistent and thus remains usable only with the single-threaded TCG.

4.2 Pinning virtual cores in QEMU

As QEMU simulates a virtual core by creating a thread, we can also talk about **thread affinity** to refers as **pinning**. Previous works show that thread affinity can have an important impact on the reduction of the execution time of parallel workloads. The

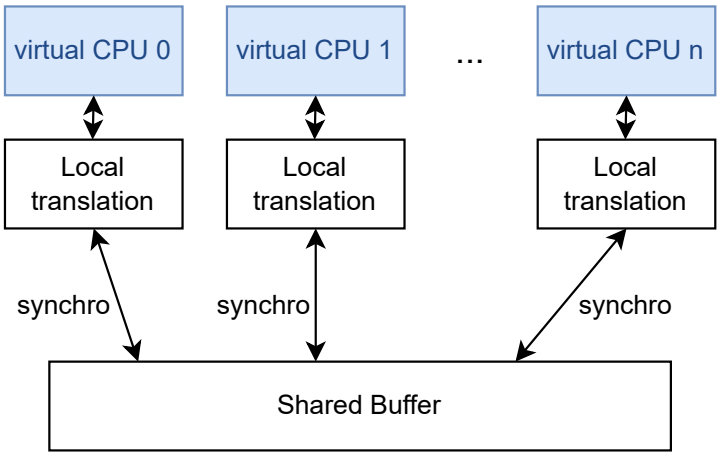


Figure 4.3: Simplified representation of the translated blocks buffer shared between the virtual CPUs

work presented in [MTB11] shows that pinning threads on High Performance Computing cache coherency NUMA machines has a real impact on the performance and thus outperformed the Linux Operating System scheduler. They have tested multiple pinning strategies using the SPEC OMP benchmark suite and all of them show improvements with HPC machines but for machines with a smaller number of cores (8 cores), improvements when pinning are marginal. Moreover, the impact of pinning is very dependent (1) on the nature of the workload, and (2) on the quality of the implementation or the parallelization of the workload. For this thesis we do not have an HPC machine but since thread affinity has been proven in past works to be beneficial and since we have a multi-core machine with more than 8 cores, pinning the virtual CPUs of QEMU is an interesting topic. As it has never been upstreamed, we decided to add in QEMU the possibility to the user to set the affinity of the virtual CPUs with the thought that it might improve the simulation time of QEMU.

4.2.1 Principle

Pinning or setting **thread affinity** is the process of forcing a thread to execute on a chosen physical CPU and it only. As seen in Figure 2.1 in Chapter 2, a core designates the entity that can be composed of one or two hardware threads (harts) depending on if the host supports the Simultaneous MultiThreading (SMT) execution paradigm. We refer to CPU as the smallest processing unit, meaning the hart. In Figure 2.1, the server can run with either 24 or 48 CPUs respectively if SMT is disabled or enabled on the machine. Thus, we use the term CPU when describing the pinning mechanism. We remind that QEMU creates a thread for each virtual CPU that executes in the simulated system.

Figure 4.4 represents a simplified example of the pinning principle when assigning 4 virtual CPUs to run on 4 physical CPUs. In this example, we arbitrary choose to run the virtual CPU 0 on the physical CPU 0, the virtual CPU 1 on the physical CPU 1, the virtual CPU 2 on the physical CPU 3 and the virtual CPU 3 on the physical CPU 2. We focus on the processor simulation scalability and as we have only a few I/O devices in the system, we will not assign a CPU for the thread that simulates these devices.

In practice, the distribution of the CPUs on the host machine is more complex. All

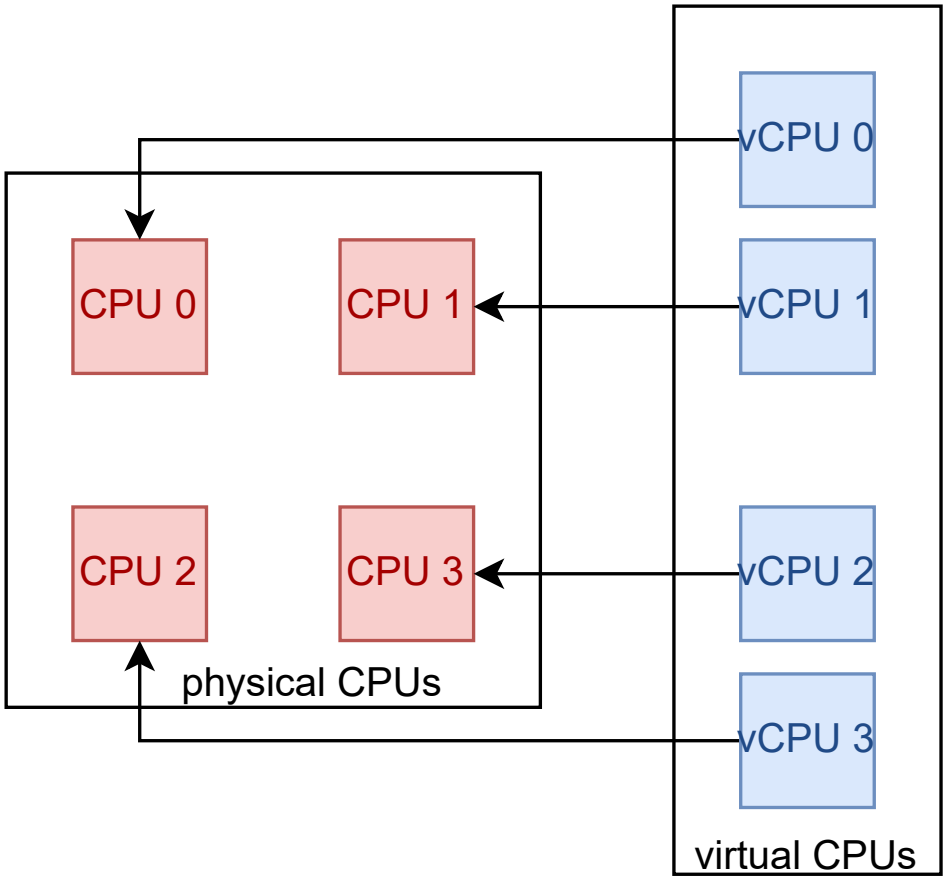


Figure 4.4: Simplified representation of the pinning mechanism

servers are now multi-core, support SMT and also have Non-Uniform Memory Access (NUMA) which means that multiple memory zones exist in the system and they are put at different places. Because of that, the pinning strategy must take into account the particularities of the host. That is what Section 4.3 is about.

4.2.2 Implementation in QEMU

Linux interfaces

The implementation in QEMU is done through the call of the Linux interfaces `cpu_set_t` and `pthread_setaffinity_np`. When a virtual CPU is initialized in QEMU, a variable of type `cpu_set_t` is used by setting a single bit in it to specify which CPU to do the affinity on. This is done by using the macro `CPU_SET`. It is thus possible to define a set of CPUs with `CPU_SET` and for this contribution the mapping was one to one. Then this variable is passed to the call of the function `pthread_setaffinity_np` that stores this information into the thread structure. Later, when the thread is scheduled, the Operating System has no other choice than to put it onto the designated CPU.

Listing 4.1 shows the part of code that does the pinning in QEMU. The code is quite small and not very intrusive in QEMU.

QEMU command line

To let the user decide on which physical CPUs he wants to do the pinning, new options in the QEMU command line need to be added.

Listing 4.1: Part of code in QEMU that implements the pinning mechanism

```
// QEMU function to initialize the virtual CPUs
void qemu_init_vcpu(CPUState *cpu)
{
    // variable of type cpu_set_t
    cpu_set_t cpuset;

    // Retrieve the affinity given in the command line
    MachineState *ms = MACHINE(qdev_get_machine());
    MachineClass *mc = MACHINE_GET_CLASS(ms);
    unsigned affinity = mc->vcpu_affinity[cpu->cpu_index];
    ...

    // If the affinity is specified in the command line
    if (affinity != -1) {
        // Put all the bit of the variable at 0
        CPU_ZERO(&cpuset);
        // Set the right bit of the variable
        CPU_SET(affinity, &cpuset);
        // Set the affinity of the thread representing the
           virtual CPU
        pthread_setaffinity_np(thread, sizeof(cpu_set_t), &cpuset
           );
    }
    ...
}
```

Listing 4.2: Pinning QEMU command line

```
qemu-system-riscv64 \
  -smp ... \
  -vcpu vcpunum=$vcpu_number,affinity=$host_physical_cpu_number \
  -vcpu vcpunum=$vcpu_number,affinity=$host_physical_cpu_number \
  -vcpu vcpunum=$vcpu_number,affinity=$host_physical_cpu_number \
  ...
```

Listing 4.2 shows what have been added to the classical QEMU command line. The `-smp` option already exists in QEMU. It is a processor related option to simulate a Symmetric MultiProcessing (SMP) system. The user can specify for the simulated target the total number of virtual CPUs, the number of cores and the number of hardware threads. Other sub options to specify the topology of the system are available but are not interesting for us. What is new is the `-vcpu` option to assign the virtual CPUs with the physical CPUs for the pinning. The user gives two numbers when using the option: one for the virtual CPU number and the other for the physical CPU number.

Listing 4.3 is an arbitrary example to show how the command line can be used. In this example we have a system with a total of 10 virtual CPUs distributed on 5 cores with 2 threads per core. If some sub options are not specified, their values are

Listing 4.3: Pinning QEMU command line arbitrary **example**

```
qemu-system-riscv64 -smp 10,cores=5,threads=2 \
-vcpu vcpunum=0,affinity=0 -vcpu vcpunum=1,affinity=8 \
-vcpu vcpunum=2,affinity=1 -vcpu vcpunum=3,affinity=9 \
-vcpu vcpunum=4,affinity=2 -vcpu vcpunum=5,affinity=10 \
-vcpu vcpunum=6,affinity=3 -vcpu vcpunum=7,affinity=11 \
-vcpu vcpunum=8,affinity=4 -vcpu vcpunum=9,affinity=12 ...
```

automatically computed. One can also take the liberty to not assign all the virtual CPUs of the target system.

4.3 Possible impacts on the scalability of QEMU Parallel Implementation

Using the pinning in QEMU might improve the simulation time as thread affinity was proven to be efficient in some works. However, in order to optimize the possible gain, the pinning must be done in a smart way. The idea is not to randomly decide the physical CPUs on which the assignment will be done. The topology of the host machine needs to be considered as well as the benchmarks used to do the experimental study.

4.3.1 Nature of the host machine

The Linux command `lstopo` shows the topology of the host machine's hardware organization. For concreteness, we assume that we do the experimental study on QEMU parallel implementation on a Dell PowerEdge R910. We give Figure 4.5 the topology of this server.

We can see that our server is divided into 4 sockets with 1 NUMA node in each socket. Then we have 4 cores per NUMA node and 2 harts per core. We have a total of 32 CPUs. Like almost all current servers, it supports the Simultaneous MultiThreading meaning that two hardware threads can run inside a core. Modern Operating Systems offer the possibility to enable and disable the SMT support during run time leaving only one "processing unit" per core. From a program point of view, it sees half or all the CPUs and does not make a distinction with or without SMT.

How to do the pinning regarding the architecture of the host

The most rational approach is to assign the virtual CPUs in such a way that they first fill up a core, then the cores of the NUMA node. We then move to the next NUMA node in the next socket when the previous node is full. If all the NUMA nodes are full but we still have virtual CPUs to assign, we start the strategy again by assigning the virtual CPU at the first hart of the first core of the first socket of the first node. If we look at Figure 4.5, it means that we will assign the first virtual CPU on Processing Unit (PU) #0, the second one on PU#16, the third one on PU#4, the 9th on PU#1 and so on with the 33rd back to PU#0. The rationale behind this strategy is that the closer the virtual CPUs on the host, the better the sharing of the resources. Indeed, the current servers predominantly only share their last level cache in a socket, the other levels being private to a core. Testing without the SMT on the host also needs to be

Figure 4.5: Mid-end host server: Dell PowerEdge R910 (**lstopo** picture)

done. It could be beneficial for the processing units to not share a core to thus only one will execute by core.

The Operating System that runs on the host machine creates kernel threads that are constantly running. The Linux kernel option `isolcpus` can specify which physical CPUs we want to exclude from scheduling. By using this option, we can make sure that the kernel threads will never run on the physical CPUs that we want to dedicate to QEMU with the pinning and thus avoid context switches that can have an impact on the simulation speed. The natural choice is to let the kernel threads execute only on Core 0, meaning that Core 1 to 15 are for QEMU.

4.3.2 Nature of the benchmarks

The choice of the benchmarks used to evaluate performances and study scalability is of primary importance. We aim at measuring the scalability of QEMU with multi-core execution and we want to do so with presumably scalable non-synthetic workloads. The PARSEC suite [BKSL08], for Princeton Application Repository for Shared-Memory Computers, is a benchmark suite of multi-threaded applications which make use of the

POSIX thread library for supporting parallelism. The programs in the suite are very diversified and they deal with classical mathematical algorithms as well as financial analytics applications and video encoding. Issues regarding the scalability of some benchmarks have been highlighted by [SR15]. However, because of the lack of a better suite and because the PARSEC suite is widely used in the field, we decided to use it anyway. Note that the older SPLASH 2 benches, references of their times, are included as is in the PARSEC suite.

As one of the goals of QEMU is to support cross-ISA simulation, we chose to cross-compiled the benchmark in RISC-V using the RISC-V GNU Compiler Toolchain with the flag `-O3`. The target is our Dell PowerEdge R910 server with an x86-64 architecture. However, not all the benchmarks are meant to be cross-compiled. Some of them fail to be cross-compiled and others that are cross-compiled successfully fail during run time due to various problems such as initialization or memory allocation. This is well documented otherwise, so we simply ignore the few programs that fail. In the end we were able to cross-compile and execute faithfully the following benchmarks: `blackscholes`, `bodytrack`, `cholesky`, `ferret`, `fft`, `fluidanimate`, `freqmine`, `lu_cb`, `lu_ncb`, `ocean_cp`, `radix`, `swaptions`, `water_nsquared` and `water_spatial`. In this subset of all benchmarks, some of them have additional constraints: `fft`, `fluidanimate`, `ocean_cp`, `radix` and `swaptions` must be executed with a number of threads that is a power of 2. Last but not least, `swaptions` cannot be executed if the number of threads is higher than internal variable swaptions which is 64.

All parallel programs have a sequential part during run time. Depending on how the benchmark is written and what needs to be initialized, the sequential part can be more or less important. The Amdahl's law [Amd67] teaches us that the sequential part of a program can have a non-negligible impact on the scalability. The law defines the theoretical speedup with the following formula $\frac{1}{(1-p)+\frac{p}{s}}$ where s is the speedup of the parallel part and p is the proportion of parallel part in the entire program. The theoretical maximum speedup is always bounded by the section of the program that cannot be parallelized. That is why it is important to measure the wall clock time of the full execution of the benchmark to do our study but also the Region-Of-Interest (ROI) part which is the execution of only the parallel section. Thankfully, the API of the PARSEC suite offers the possibility to retrieve the execution time of the ROI for each benchmark.

Figure 4.6.(a) shows the classical structure of the PARSEC benchmarks at a high level of abstraction. First, we have the sequential part that can be the initialization of data structures, memory allocations, parsing of files and so on. Then the ROI timer starts and the threads begin to execute. The threads are more or less independent, it depends on the shared variables present in the benchmarks. They can induce waiting time as they are protected by locks and thus also have different interleavings if run several times. At the end of the parallel execution, the threads are synchronized at a barrier and can start again to compute if necessary. The ROI timer is then deactivated and the programs exist. Figure 4.6.(b) is for QEMU's view of the target. Each virtual CPU on which the PARSEC threads will execute can potentially access the whole memory and synchronization will happen when instructions perform atomic operations and fences.

Another interesting feature of the PARSEC suite is its ability to set thread affinity which is perfectly in line with our needs. Each program in the suite takes as argument

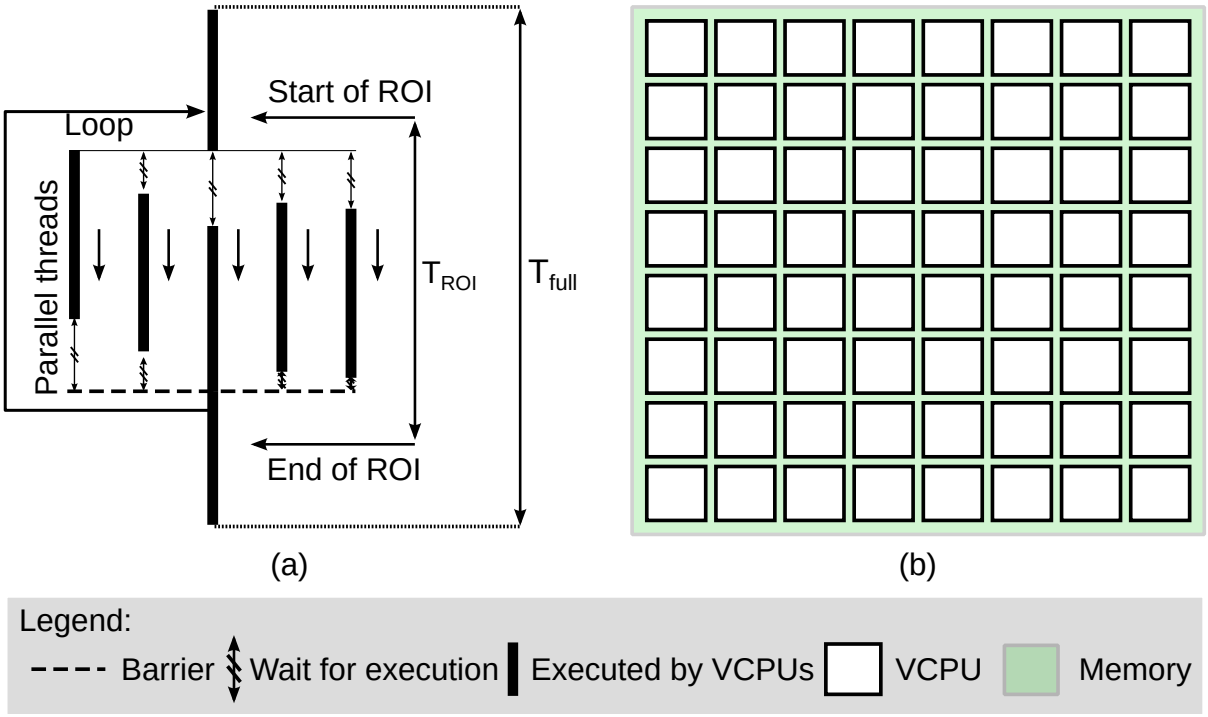


Figure 4.6: (a) Parallel programs structure and (b) Target architecture organization

the number of threads to execute the program on. By using environment variables, it is possible to set the affinity of the threads. In our study, the idea is to use the PARSEC thread affinity on top of QEMU. The threads will be assigned to the virtual CPUs of QEMU. The API lets the user set two environment variables: `PARSEC_CPU_NUM` and `PARSEC_CPU_BASE`. The first one is to specify the maximum number of physical CPUs to use and the second one is to give the id of the base CPU to use meaning that the CPUs below that id will not be used. As a result, these variables can define the range of CPU ids in which the PARSEC threads will run. The naive way of setting these variables is to do: `PARSEC_CPU_BASE`=0 and `PARSEC_CPU_NUM`=n the number of threads of the benchmarks meaning that the benchmarks will run on the n first CPUs on the host. This is the configuration we adopted for our study.

4.3.3 Simulation Methodology

Here is a summary of all the parameters in our experimental study:

- number of host hardware threads (harts) per core which can be 1 or 2 (Simultaneous MultiThreading enabled or not). It leads us to 16 or 32 CPUs on the host machine,
- number of virtual CPUs that we will call n_c. This number will take the following values: $\{1, 2, 4, 8, 16, 24, 32, 48, 64, 96, 128\}$,
- number of threads n_t that the PARSEC benchmarks will generate during execution. This number will take the following values: $\{1, 2, 4, 8, 16, 24, 32, 48, 64, 96, 128\}$,
- PARSEC threads affinity,
- pinning QEMU virtual CPUs to physical CPUs,

- `isolcpus` to strictly separate the physical CPUs allocation between QEMU virtual CPU threads and the kernel threads.

To run the PARSEC benchmarks inside QEMU, we rely on busybear [JC17] which is a tiny RISC-V Linux root filesystem image specific to run with QEMU. It is composed of a Linux and OpenSBI, the official RISC-V "bios". We thus run QEMU with the busybear image that contains the PARSEC binaries.

Figure 4.7: Screenshot of busybear shell run with QEMU

Figure 4.7 shows the shell of busybear once run with QEMU. One can run the PARSEC programs that he wants (example of `radix` command line in the Figure). We use the Unix `time` command inside busybear to measure the execution time of the programs. Linux `time` accesses the host real-time clock under the hood (QEMU implementation of the RISC-V `mtime` register). We thus can be confident that we will retrieve the right execution time. In addition, the Linux timer rate heavily impacts the boot time of the virtual CPUs. As in QEMU the timer is synchronized with the host real-time clock, the slower the boot the higher the number of timer interrupts. So, we opted for a value of 100 Hz, classical for server workloads.

We made all the measures on the R910/0P658H Dell PowerEdge server presented Figure 4.5 and detailed in [GFKW11]. It is composed of 4 Nehalem-EX 7520 Intel Xeon Processors with 4 cores with 2 harts in each that results in a total of 16 cores/32 harts. We ensure only the base Linux kernel, a ssh connection and QEMU are running on the server while performing the experiments. To do an objective analysis, we also decided to force the frequency to be and remain the same for all the physical cores on the server, which is not the case by default. We force the frequency of the physical cores to be 1862 MHz through the appropriate BIOS configuration of the server.

With all the parameters that we consider in our study, multiple combinations can be done. Regarding n_c and n_t, the classical idea would be to set $n_c = n_t$ meaning we increase at the same time the number of PARSEC threads with the number of virtual CPUs. Algorithm 4.1 presents the simplified version of the basic algorithm to run the experiments.

Algorithm 4.1 Basic measurement algorithm

Require: SMT enable (two harts per core)
1: **for** n_c in {1, 2, 4, 8, 16, 24, 32, 48, 64, 96, 128} **do**
2: Set the pinning in the QEMU command line
3: Run QEMU with busybear with -smp n_c
4: Run the PARSEC benchmarks in busybear in QEMU with $n_t = n_c$
5: Retrieve the execution time (full and ROI) of the benchmarks
6: Quit QEMU
7: **end for**

Multiple variations of the parameters are thus done from this algorithm: enable or disable SMT, with or without QEMU pinning, set PARSEC threads affinity, set $n_c = 128$

and increase n_t, set `isolcpus`, etc. Section 6.2 will give more details on the multiple combinations of the parameters.

4.4 Conclusion

Given the popularity of multi and many-core systems, simulation engines adapted their strategy a few years ago to follow the trend and support parallel execution. Dynamic Binary Translation might well be the unchallenged technology for full-system level simulation, and the translator QEMU has become extremely popular. To the best of our knowledge, the scalability of the parallel implementation of QEMU was never studied before. Thus, assessing the scalability of such a widely used simulator makes sense. This chapter proposes a detailed and complete experimental study of QEMU parallel implementation with in addition the possibility to pin the virtual CPUs of QEMU on the host physical CPUs. The details of all the experiments done regarding the variation of the parameters in our study are given in Section 6.2 and this leads us to conclude on the usefulness of to pin or not to pin.

Chapter 5

Fast Cache Simulation For The Dynamic Binary Translation Mechanism

In this chapter we answer the questions "**Can we benefit from the DBT approach to design in particular a cache model that limits the impact on the global simulation time?**" and "**How can we enhance the time accuracy of the DBT mechanism when adding models of new architectural features in the simulation without overly degrading simulation speed?**". Here again we chose QEMU as the DBT-based engine and we rely on the QEMU TCG Plugins as the support to implement a cache simulation. As the simulation with the DBT is purely functional, our goal for this contribution is to produce a cache simulation in a smart way inside QEMU at a high level of abstraction. Instrumenting each target instruction to simulate a cache is achievable but it will degrade the performance a lot. By relying on the per instruction block execution of the DBT principle, we propose an instruction cache model at that granularity that is faster than a per instruction instrumentation.

This chapter presents the initial intuition behind our per block instruction cache simulation that helps to reduce the overhead of adding a new simulation model. We also explore other solutions to propose a fast cache simulation in addition to the improvement for the instruction cache. However, this raises a few issues that we detail and mitigate.

Table of contents

5.1 Introduction

As introduced in Chapter 2, a cache is a small memory that holds copies of data that otherwise reside in main memory and are now presents in each and every processor one can think of. Usually, multiple levels of caches are present in a memory system. This is called cache or memory hierarchy. First, we have the Level 1 (L1) cache that is most of the time split into two independent parts: L1i to store the instructions and L1d to store the data. This cache level provides the faster access time but is small (from a few kilobytes to max 32kB even on high end processors). Then we have the L2 cache which is bigger and slower and finally depending on the architecture we can have a last level that is shared between multiple CPUs (called L3 or LLC for Last Level Cache). Figure 5.1 illustrates an arbitrary cache hierarchy.

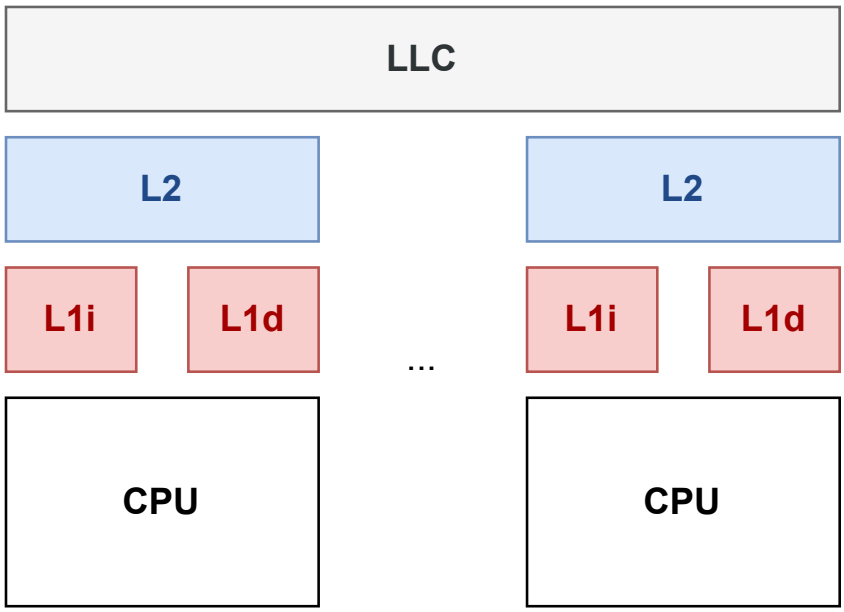


Figure 5.1: Arbitrary example of the cache hierarchy

Our contribution focuses on the model of a L1i + L1d + L2 per CPU. We will not address the model of a LLC shared between multiple CPUs.

When doing a cache simulation at a high level, only the directory holding the tags and a bit indicating if the tag at a given index is valid or not are necessary. This information is sufficient to produce statistics on the number of hit/miss and number of evictions. The index of the line 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 translation of blocks in QEMU occurs with a known virtual to physical address translation context, virtually addressed, physically addressed or virtually indexed physically tagged caches can be simulated.

5.2 Instruction cache modeling: L1i

Said so, instruction cache simulation is straightforward. It is, however, slow because we need to simulate each and every instruction address. We therefore propose a coarser grain simulation to enhance execution speed.

5.2.1 Initial Intuition

Our initial intuition relies on the TB per TB execution principle of the DBT. As explained in Figure 2.5, blocks of instruction are translated instead of a translation per instruction. The TBs are built in such a way that the last instruction in the block is an instruction that will jump to another address. With this information, we have the warranty that once we have entered a TB, all subsequent instructions within the TB are at consecutive addresses. Thus, we can know for sure which instructions will hit and which might miss. When entering the TB, we must verify if the first instruction misses as we do not know in advance if it is a hit or a miss. However, from now on, the following instructions that hold on the same cache line, *i.e.* that share the same index, will hit for sure. When we arrive in the TB on an instruction that belongs to another cache line, we must check if the instruction misses but here again we will have a hit for the instructions that follow. As a result, the consecutive addresses concept inside a TB helps us to have a prior knowledge of the instructions that will be a hit for sure and of the instructions that might be a miss. In the end, the cache simulation is necessary on only the unsure instructions and we can detect these instructions during the creation of the TB. Figure 5.2 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 5.2: Example of static hit/miss decision within a TB

We represent here a 16-byte cache line, *e.g.* the line base addresses have their 4 least significant bits zeroed. The instructions are from the RISC-V ISA (they are a mix of compressed, 16-bit, and normal, 32-bits instructions). When we enter the TB, we have the address `0x800fa7bc` that lays in the middle of a cache line. We have to check if this address will induce a hit or a miss but as the next address `0x800fa7be` is on the same cache line (illustrated in blue on the Figure), we know that this next address will be a hit. Then, the next instruction at the address `0x800fa7c0` is on another line (illustrated in yellow on the Figure). Thus, we need to determine if this address will cause a hit or a miss. However, the following 4 instructions will be for sure a hit since they belong to the same line. Overall, we only need to run the cache simulation on 2 addresses in the TB compared to 7 instructions when doing a naive cache simulation. Consequently, we can drastically reduce the overhead of doing cache simulation.

5.2.2 Proof of concept

We have just seen the intuition on which we rely to create a smart L1i model. In order for the model to work in any circumstances (in theory), some assumptions need to be fulfilled. The TB is determined after the translation stage, so at this step in the DBT mechanism, we need to know the following information:

- the number of instructions in the TB,
- the address of each instruction in the TB

Fortunately, QEMU gives us what we need to know so we are sure to be able to make our model works.

Figure 5.3: Instruction address split

The last point to be clarified concerns how to know on which lines of the cache the instructions will belong. Figure 5.3 shows that the instruction address is split into three parts. The size of each part (number of bits in the address) depends on the cache configuration. The `offset` is fixed by the number of bytes on each cache line and is computed by doing $log2(bytes\_per\_line)$. The `index` corresponds to the number of line sets in the cache and its size is computed by doing $log2(sets)$. In the simplified example Figure 2.10 we had n sets and in each set we have 2 ways. We can deduce the `tag` size thanks to the previous ones. Therefore, two addresses that hold on the same cache line must have the same [**tag + index**] part. Retrieving this part of each address is straightforward as it requires doing a simple mask.

5.2.3 Implementation

We did the implementation of our L1i model with the QEMU TCG Plugins. We showed Figure 3.2 the mechanism of these plugins and we illustrated Figure 3.3 the existing naive cache plugin that does the simulation of a L1i, L1d and L2 per virtual CPU. In this section, we focus on the implementation of our L1i model and how it differs from the L1i in the existing cache plugin.

Cache Simulation at TB Granularity

The API of the QEMU TCG Plugins is limited but it offers the possibility to retrieve the target instructions inside a TB during the translation stage. All the existing plugins have the same pattern. The entry point is the `qemu_plugin_install` function that each plugin needs to write. In this function, the first step is to register to the TB translation through the callback function `qemu_plugin_register_vcpu_tb_trans_cb`. Then at this point we have two options:

- The first one implemented by the existing cache plugin 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 are known.

– The second one implemented by our plugin is to register a callback each time a TB executes. During the TB translation stage, the API of the Plugins provides a way to parse the TB and have access to the instructions it contains. We store in an array the list of the instructions that might potentially be a miss and we give this array as argument to the TB execution callback function. The L1i simulation is run on the possible misses in the array at once at the beginning of the TB execution.

Figure 5.4 sums up the mechanism of the L1i of the existing cache plugin. This plugin subscribes to the events ”instruction execution” which means that each time a target instruction executes, the L1i simulation is run. Doing the L1i simulation at the instruction granularity thus induces a non-negligible overhead. Figure 5.5 shows how we do our L1i differently than the existing one. Unlike the existing cache plugin, we subscribe to the event ”TB execution”, which occurs much less often. The number of instructions per TB is very dependent on the nature of the application. The commonly stated average number of instructions per block is less than 10. However, for applications that contain algorithms with lots of loops (which is the case of the benchmark suites that we used), this number can be higher. Moreover, with such benchmarks, the percentage of TB reuse is really high, meaning that the overhead in our solution of storing an array for the list of potential misses is amortized.

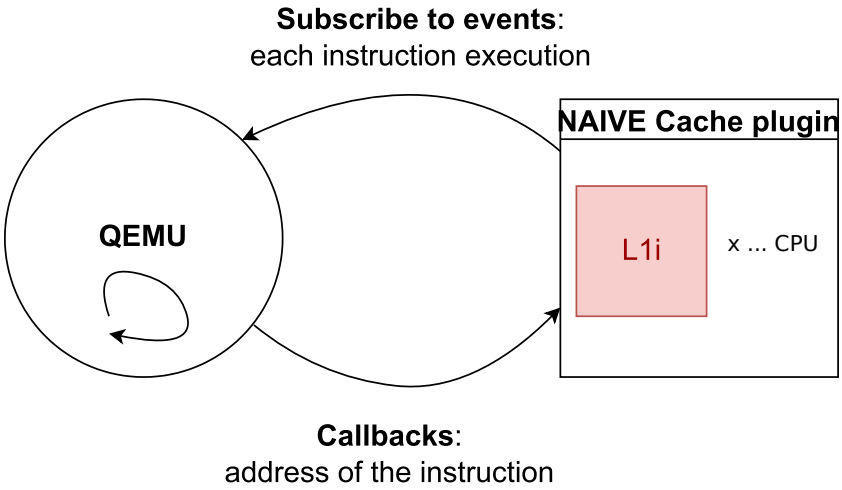


Figure 5.4: Simplified representation of the L1i implementation in the **existing naive cache plugin**

Figure 5.6 shows the differences of callbacks insertion in the TBs for the existing L1i cache plugin and our L1i cache plugin. We can see that our version produces only one call compared to the existing plugin that generates a call for each instruction. In our version, this call is the one that performs the L1i simulation on the sequence of potential instruction misses. When we record the potential misses in an array at the translation stage, we use dynamic memory allocation and it can be a bottleneck. However, thanks to the DBT mechanism, the TBs are put in a cache to be reused. The TBs are reused at a high percentage (depending on the application but is true for a majority) so the impact of using dynamic memory allocation for each TB is really negligible.

Cache characteristics

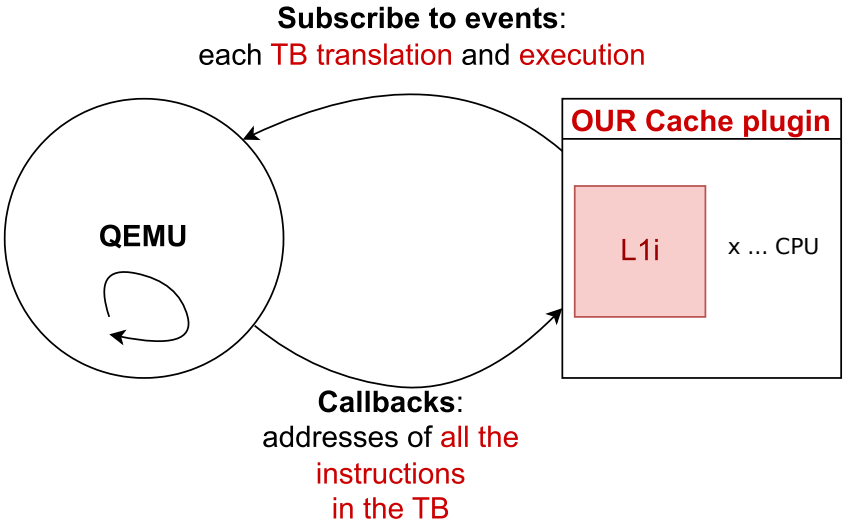


Figure 5.5: Simplified representation of the L1i implementation in **our cache plugin**

```
# NAIVE solution                          # OUR solution
CALL_PLUGIN_ins(...)                      CALL_PLUGIN_tb(...)
0x7f1ffbe5693e :   sb       s0,1470(a5)   0x7f1ffbe5693e :   sb       s0,1470(a5)
CALL_PLUGIN_ins(...)                      0x7f1ffbe56942 :   ld       s0,0(sp)
0x7f1ffbe56942 :   ld       s0,0(sp)      0x7f1ffbe56944 :   addi     sp,sp,16
CALL_PLUGIN_ins(...)                      0x7f1ffbe56946 :   ret
0x7f1ffbe56944 :   addi     sp,sp,16
CALL_PLUGIN_ins(...)
0x7f1ffbe56946 :   ret
```

Figure 5.6: QEMU TCG Plugins callbacks example on a TB

The characteristics of a cache are defined by its size (number of ways, sets and bytes per line), the replacement policy and the writing policy if the cache holds data that can be read or written. In the case of the L1i model, we do not need a writing policy as the instruction cache only stores addresses of instructions to be read. Both the existing cache plugin and our plugin offer the possibility to set a cache of any size. The following Table 5.1 sums up what are supported for the replacement policy in the existing cache plugin and in our plugin for the L1i model. We use **cache** to refer to the existing cache plugin and **cacheTB** to refer to our plugin.

	Random	LRU (Least Recently Used)	FIFO (First In First Out)	1-bit
cache	yes	yes	yes	no
cacheTB	yes	yes	yes	yes

Table 5.1: Sums up of the Replacement Policies supported by the existing cache plugin and our plugin

We took the liberty to implement another replacement policy that is not present in the existing cache plugin. We implemented the 1-bit policy which is a pseudo LRU policy. A status bit is used for each set in the cache. These bits are named MRU for Most Recently Used. When a place needs to be freed in a given set, multiple ways are possible. With the 1-bit policy, if the bit in the set is 1, we evict randomly a way in the

first half of the ways and in the second half if 0. When access to a given set and way is done, the bit is updated to 0 if the way belongs to the first half of the ways and 1 if in the second half. This policy is quite efficient for its cost and has been used in ARM processors, as presented in Damien Gille's Master Thesis [Gil07].

5.2.4 Error in Counting Instructions and How to Mitigate it

Our L1i model works well in theory but some errors in counting instructions do happen in practice. With our model, we compute in advance, during the translation stage, which instructions might miss. When doing so, we make the assumption that all the instructions in the TB will be effectively executed. However, this is not what happens in reality and thus we wrongly count the number of instructions. Alas, this problem only happens with our L1i and does not affect the existing cache plugin as it relies on the callback for the execution of each instruction.

The source of this counting error are the exceptions. 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.

```
# Insns in translation block
CALL_PLUGIN_tb(...) # 9 insns counted
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)
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)

# New translation block after return from handler
CALL_PLUGIN_tb(...) # 7 insns counted
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
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)

# New translation block after return from handler
CALL_PLUGIN_tb(...) # 4 insns counted
0x7f1ffbe5693e :   sb        s0,1470(a5)
0x7f1ffbe56942 :   ld        s0,0(sp)
0x7f1ffbe56944 :   addi      sp,sp,16
0x7f1ffbe56946 :   ret
```

Figure 5.7: Stopped TB execution example due to a page-fault

Figure 5.7 illustrates this problem with page-faults that occurs once the kernel page table has been set up during the Linux boot. On the left side of the Figure, we have

a TB freshly translated. We counted 9 instructions. During the execution of this TB (beginning of the right side of the Figure), the first store in purple provokes a page-fault. Because of that, a new TB is created after return from handler on which we count 7 instructions that were already counted in our model in the original TB. Then the execution continues and we arrive at a second store in purple that also provokes a page-fault and thus the creation of another TB with 4 instructions already counted in the original TB.

Other instructions than the page-fault one produce this error such as `wfi` (that sleeps waiting for an interrupt) or `pause` (that yields back the processor). But unlike the page-fault they are easy to handle. They happen very rarely and when we force to end the TB, the slowdown is insignificant. However, regarding the memory accesses, they occur very more often. It depends on the workload but in general 30% to 50% of the instructions are memory accesses. So, ending the TB on all loads and stores just because that might raise a page-fault can induce significant slowdowns. Section 6.3 evaluates in detail the real impact of this problem on the cache statistics.

5.2.5 Dependency on Simulator Runtime

We also discovered another unexpected behavior for programs running on top of Linux, and that does not show up for bare metal programs. We came across a time dependency of the flow of executed target instructions. We brought out this odd phenomenon while testing several cache implementations. Surprisingly, the faster the simulator, the lower the number of executed instructions for a given program. As far as we are aware, this phenomenon has not yet been reported in the literature. Doing the evaluation with such an effect is not wrong in essence because the instructions executed induce a correct behavior but can make previous works questionable regarding the cache statistics they report. For example, a simple direct mapped cache would lead to a smaller number of executed instructions and possibly better hit rate than a fully associate cache, just because the latter is much slower to simulate than the former.

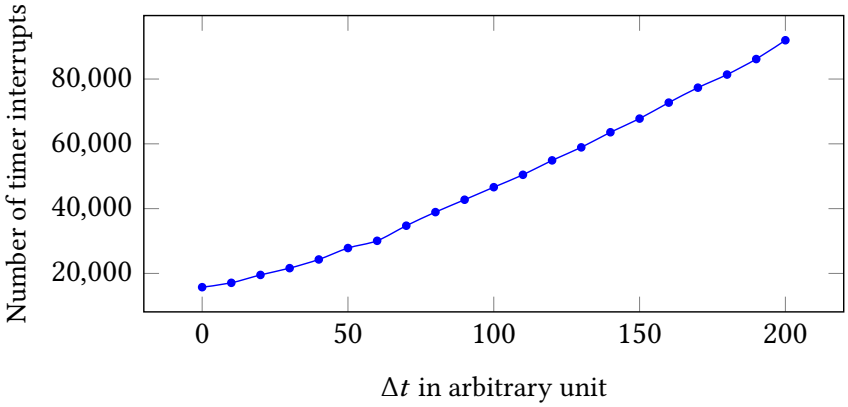


Figure 5.8: Number of raised timer interrupts as a function of an arbitrary cache model simulation delay

When investigating why the behavior happens, we discovered that it comes from the repeated occurrences of timer interrupts. QEMU triggers alarms by relying on the host real-time clock (using rdtsc on x86 hosts). At each Δt time elapsed, the timer interruption is raised and a call to the target Linux scheduling function `update_cfs_group` is done. Calling this function generates instructions that will be taken into account by

the TCG Plugins. So, if a plugin does a lot of computations, the timer interruption will be raised a lot and thus a lot more instructions will be executed. We illustrate this in Figure 5.8 by adding a for loop of n iterations in our cache model to induce a delay. As a result, the more interrupts, the more instructions counted in the cache statistics. Unlike the problem above, this dependency on the simulator runtime affects both our plugin and the existing cache plugin.

This dependency effect can be reduced by using the `-icount shift=2,sleep=on` option of QEMU. It forces QEMU to use an internal clock in which case only 1830 irqs are raised, for any added delay. However, this option of QEMU does not support multi-core execution of the virtual CPUs. As we want to make a fair comparison of speed with the existing cache plugin and with vanilla QEMU, we use either bare metal software that does not program alarms or QEMU `user`-mode to do the evaluation. They both support parallel execution and are not affected by the time dependency effect. Another solution could be to simulate the instruction cache in parallel of the execution of the main loop of QEMU, in a separate thread by writing in a FIFO. But this solution needs to be mitigated as even if it can solve the dependency on the simulator runtime, it induces other issues that are addressed in the next section with the L1d model.

5.3 What about other cache levels: L1d and L2?

Thanks to the TB per TB execution of the DBT principle, we propose a L1i model which allows to reduce the number of calls to the cache simulator by knowing at the translation stage which instructions will hit for sure and which will might miss. Thinking of a similar model for the L1d is more complicated as the addresses of the memory accesses are not necessarily consecutive. Thus, to improve the performance compared to a naive L1d (which runs the cache simulation each time a memory access executes), we propose a threaded execution of the L1d model. Regarding the L2 model, the question is raised on how we can link our two different models of the L1i and L1d to produce the L2 simulation per virtual CPU.

5.3.1 Threaded L1d simulation

The principle of our L1d model is that the memory accesses are retrieved thanks to the callback function in the TCG Plugins API and are put in buffers whose number and size are adjustable. Once a buffer is full, it is sent to the thread that does the cache simulation at once on all the memory accesses in the buffer. Figure 5.9 illustrates this process. We have in gray what corresponds to the same thread, similar to the Figure 5.4 and Figure 5.5 and in red the thread that does the data cache simulation. However, in the case of multi-core simulation, multiple ”gray” threads will be created and each will correspond to a virtual CPU. Thus, multiple threads will communicate with the ”red” thread thanks to synchronizations.

Figure 5.10 shows more in details how the buffers are fill up and how they are sent to the cache thread. We allocate at the beginning a list of available buffers for each virtual CPUs: `free_buffers`. Each virtual CPU starts by taking a buffer in its list. The buffer that is being filled is called the `current_buffer`. Once a virtual CPU has its `current_buffer` full, the buffer is added to a global list common to all threads: `full_buffers`. The global list of full buffers is a lock-free structure. While the list of

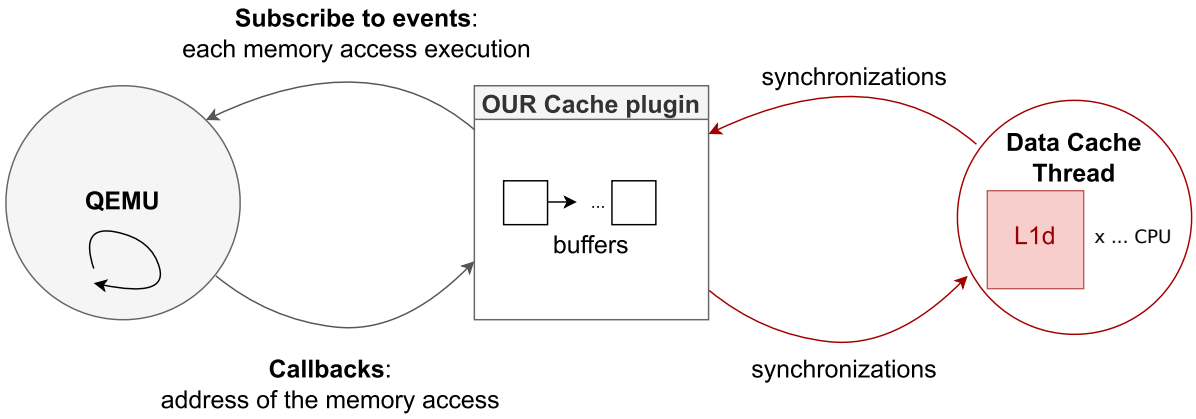


Figure 5.9: Simplified representation of the L1d implementation in **our cache plugin**

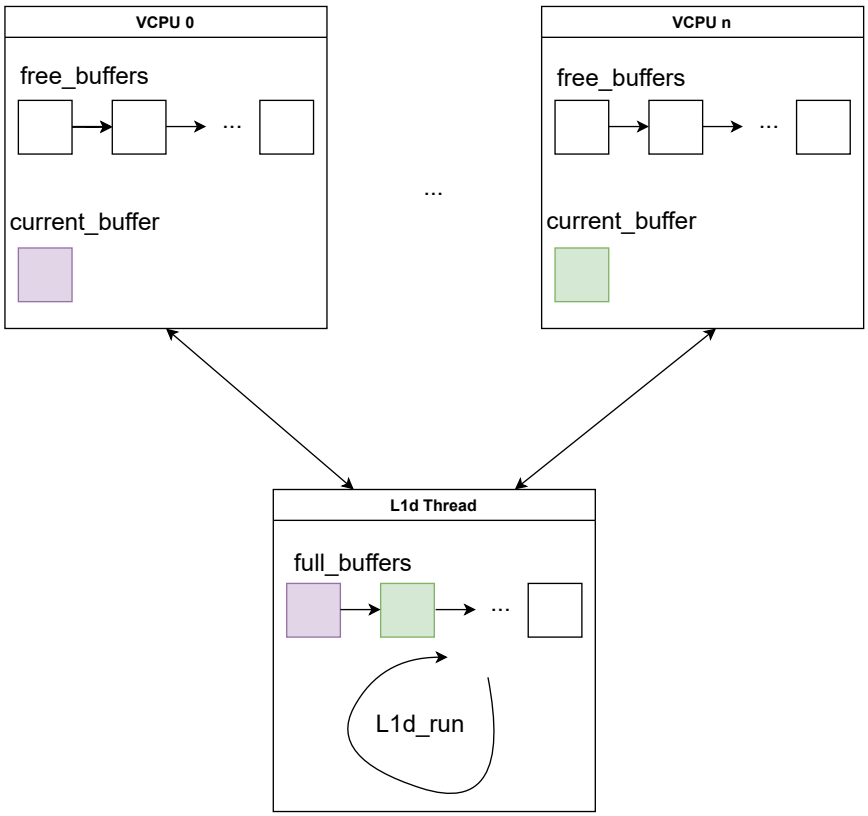


Figure 5.10: Simplified representation of the data thread interactions with the virtual CPUs

full buffers is not empty, the cache thread will run the L1d simulation on each memory access on the buffer that belongs to a virtual CPU. Once the buffer is completely consumed, it is put back in the list of available buffers. The list of free buffers of each virtual CPU is managed as a producer-consumer with semaphores. As the data cache simulation for each virtual CPU is done on a separate thread, it is out-of-sync from QEMU execution.

Remarks on the choice of the shared structure

We went through several types of structures before deciding to take the lock-free one for the list of full buffers. Our very first solution was to use semaphores to deal with this list. It was our reference implementation.

At the start of our investigation, we tested the asynchronous queue structure `GAsyncQueue` written in C and available within by the GLib library [AAG+16]. With this solution, the list of full buffers is replaced by an asynchronous queue. Moreover, the buffers in the queue can be ordered to simulate a FIFO policy. But compared to the classical list of full buffers that is synchronized with semaphores, this solution did not show better results, so we did not choose it.

After that, we were wondering if the use of semaphores can be beaten by condition variables. Table 5.2 shows the range of time for a couple of executions of the PARSEC benchmark `lu_cb` for 4 vCPUs. We have tested arbitrary values for the combination $number\_of\_buffers \times size\_each\_buffers$.

Time in s	16x8	128x128	1024x1024	1024x4096
condition	[17,7-17,9]	[6,0-6,7]	[5,2-5,5]	[6,5-7,3]
semaphore	[16,4-19]	[4,8-5,4]	[3,9-4,6]	[6,9-7,3]

Table 5.2: Execution time in `full_system` of the PARSEC benchmark `lu_cb` for **4** vCPUs

We also observed the same behavior with other PARSEC benchmarks. As one can see on the Table, using conditions instead of semaphores does not improve the simulation time and as a result we did not choose it either.

Then, our final idea was to use a lock-free structure. Thirty years ago, Maurice Herlihy presented the benefits of using lock-free structures [HM93, Her93]. Since then, his implementation was criticized ([Bar93]) but the lock-free ideology remains used in shared data structures research. For our investigation, we used a lock-free structure proposed by the library liblfds. This library contains around ten structures and we chose the ”bounded queue” one which supports many producers and many consumers. As our previous experiments showed the best performance for 1024x1024, we focused here on testing the behavior of the lock-free structure by varying the number of virtual CPUs. Table 5.3 sums up the range of times of a few executions for 2,4,8,16,32 and 64 virtual CPUs. The execution times are different from the previous Table as it was executed on a different machine.

Time in s	2	4	8	16	32	64
semaphore	[8,9-10,2]	[9,6-10,4]	[7,5-8,2]	[5,7-6,4]	[5-5,5]	[7,3-8,4]
lock-free	[8,2-9,7]	[9,1-9,4]	[7,2-7,8]	[5,5-5,8]	[5-5,5]	[6,8-7,7]

Table 5.3: Execution time in `full_system` of the PARSEC benchmark `lu_cb` with **1024** buffers of a size **1024**

The improvement when using a lock-free structure for the list of full buffers instead of the classical list with semaphore is not very significant but visible enough to make us choose this kind of structure for our implementation.

Cache characteristics

The following Table 5.4 and Table 5.5 sum up the replacement policies and the writing policies available in our plugin and with the existing cache plugin for the L1d model. **cache** refers to the existing cache plugin and **cacheTB** refers to our plugin. Compared

to the existing cache, we support for the L1d model an extra replacement policy which is the 1-bit LRU and an extra writing policy which is the Write Through.

	Random	LRU (Least Recently Used)	FIFO (First In First Out)	1-bit
cache	yes	yes	yes	no
cacheTB	yes	yes	yes	yes

Table 5.4: Sums up of the Replacement Policies supported by the existing cache plugin and our plugin

	WRITE THROUGH	WRITE BACK ALLOCATE
cache	no	yes
cacheTB	yes	yes

Table 5.5: Sums up of the Writing Policies supported by the existing cache plugin and our plugin

5.3.2 L2 simulation

Mitigations on the L1d threaded simulation

Doing the L1d simulation on a separate thread helps in relieving the plugin contention for the simulation on each memory access. It results with a L1d simulation that is done out-of-sync from QEMU main execution. This raises two issues that can have effects on a L2 simulation:

- The first one concerns the scalability of our model. For a few numbers of virtual CPUs, our threaded model might show better performance than the naive per memory access simulation. However, when we execute a high number of virtual CPUs, a bottleneck can happen since multiple virtual CPUs will want to access the same and only cache thread. Thus, it can degrade the performance as the virtual CPUs may be pending a lot. One can think of a solution to avoid this bottleneck: having a data cache thread per virtual CPU. But this also leads to some issues. If we create a thread per virtual CPU for the data cache simulation, we limit the total number of virtual CPUs that can be simulated. Imagine having a host machine composed of 128 CPUs and simulating 64 virtual CPUs with QEMU. We will have 64 other threads for the data cache simulation and results with a total of 128 threads. If we want to simulate a number of virtual CPUs higher than 64 in this example, we will have multiple threads executing on the same physical CPUs and thus degrading the performance. With only one data cache thread for all the virtual CPUs, we will have only a total of 65 threads. Moreover, if we accept this virtual CPUs number limitation, the problem is just postponed to the L2 implementation. Indeed, we will need to do synchronizations between all the data cache threads to access the L2 and thus creating a bottleneck.
- The second one concerns the out-of-sync simulation. Implementing a L2 means that we have a cache level to unify the L1i and L1d. To have a correct L2 simulation, it needs to be done at a small granularity. But having the memory accesses

simulated out-of-sync compared to QEMU main execution will have an impact on the validity of the data contained in the L2. A solution could be to do the L1d+L1i+L2 simulation on a separate thread but we come back to the first issue regarding the scalability in multi-core.

Section 6.3 highlights these issues on the scalability and conclude that only a small number of virtual CPUs show improvements.

L2 implementation

To have a correct simulation of the L2, we decided to keep our L1i that relies on the TB per TB execution and we also keep the naive L1d simulation at each memory access execution (with the callback of the TCG Plugins API for the memory access). Each time a memory access executes, the L1d simulation is run as well as the L2 simulation. Each time a TB executes, the L1i simulation is run on each possible miss of the list retrieved during the translation stage as well as the L2 simulation. Table 5.6 sums up the inclusion policies supported by our plugin and by the existing cache plugin.

	INCLUSIVE	NINE (NON INCLUSIVE NON EXCLUSIVE)
cache	no	yes
cacheTB	yes	yes

Table 5.6: Sums up of the Inclusion Policies supported by the existing cache plugin and our plugin

5.4 Conclusion

Thanks to the DBT mechanism, DBT based engines such as QEMU propose a fast and functional simulation. Adding instrumentation in functional simulators will for sure degrade the performance. In this chapter, we proposed diverse ways to minimize the performance overhead induced by adding a cache simulation inside QEMU. By relying on the QEMU TCG Plugins to implement our cache hierarchy model per virtual CPUs, we make sure to not be intrusive in the QEMU source code and thus this makes our implementation sustainable over time as the source code evolves.

By taking benefit of the per TB nature of the translation in the DBT, we defined a strategy to model the instructions caches that considerably reduces the overhead caused by instrumentation. We however raised two issues, one due to the DBT mechanism and how we thought of our L1i model and the other independent of our model related to the time dependency.

The same intuition was more complicated to adapt for the data cache. In order to still reduce the overhead of instrumentation, we proposed a threaded data cache simulation. This contribution however raises questions on the scalability when lots of CPUs are simulated and how to assert the correctness of the L2 simulation since the L1d simulation is out-of-sync from QEMU main execution.

Overall, fast cache simulation is not that easy and one of the main issue remains on how to assert the validity of the produced metrics which will be addressed Chapter 6.

Chapter 6

Experiments

Up to now we have seen different ways to improve the DBT engines simulation time and how to design smartly a cache model thanks to instrumentation that limits the overhead on the global simulation time. Thanks to the development and the popularity of multi-core systems, the DBT engines adapted their designs to propose a virtual multi-core simulation that relies on the parallel architecture of the host. As multi-core execution on the host is in itself an improvement to accelerate the execution time of softwares, parallel DBT engines that use the parallel architecture of the host also provide good performance. As it has never been done to the best of our knowledge, we proposed in Chapter 4 a methodology to study the scalability of the parallel implementation of the well-known DBT engine QEMU. We also presented a methodology to pin the virtual CPUs of QEMU on the physical CPUs of the host machine with the idea that it might improve the performance. In Chapter 5 we focused on how to add new architectural features in the simulation without degrading a lot the performance. Thus, we contributed to efficiently model configurable caches in QEMU. We proposed an instruction cache simulation that takes benefit of the per TB execution of the DBT mechanism and that considerably reduces the overhead of adding a cache simulator inside QEMU. However, this model raised few issues that are studied in this current chapter. We also proposed other simulation solutions for the data cache model.

In this chapter we firstly detail the experiments environment with all the benchmarks that we used. Secondly, we present the results of our methodology on the scalability of QEMU parallel implementation and we conclude on the impact of the pinning on the simulation performance. Thirdly, we present the performance results of our cache simulation in QEMU and we focus on how we validate those results by comparing them to another cache simulator. Fourthly, we combine our two contributions.

Table of contents

6.1 Environment

This section describes the environment that we used to do the experiments. It is composed of the host machines on which we run QEMU, which QEMU version we used, which ISA we chose as the target in QEMU, which Operating System we cross-compiled to be run in QEMU and finally with which benchmarks we do the evaluation.

Host machines

We relied on two servers to run our experiments. The first one was mentioned Figure 4.5. This is the server on which we did the analysis of QEMU parallel implementation scalability with and without the pinning. It is a Dell PowerEdge R910/0P658H. It is composed of 4 Nehalem-EX 7520 Intel Xeon Processors with on each 4 cores (4x4 cores with 2 hardware threads per core that results with a total of 32 CPUs). During the second year of this thesis, we acquired a new server with more cores than in the previous one. It is on this new server that we run our experiments to evaluate our second contribution Chapter 5 of our cache model. This new server is a 2 GHz 64-core AMD EPYC 7702P PowerEdge R6515 server, with 128 CPUs in total. On both the two machines, we were the only user connected through a ssh connection and the host Operating System is Debian 11.

QEMU

As QEMU is open-source, we retrieved the source code on the public git and we compiled it. Regarding the first contribution, we did an analysis with the version of two years ago. Since then, work has been done manually to make the pinning implementation up to date with the latest version of QEMU. Concerning the second contribution, as we relied on the TCG Plugins, our cache implementation is viable with the latest versions of QEMU (plugins are available since QEMU V4.2, but their API is still evolving).

Target ISA

The boost around the RISC-V ISA in academic and industries made us choose this ISA as the target in QEMU for our 2 contributions. To make a complete analysis of QEMU scalability, we also did the scalability evaluation with ARM as the target ISA. Regarding the second contribution, as our cache implementation is not ISA dependent thanks to the TCG Plugins, we run only the experiments with a RISC-V target.

Busybear Linux

Busybear Linux is a RISC-V root filesystem image comprised of busybox and dropbear. When we did the experiments for our first contribution, we used linux-5.9.6, dropbear-2020.81 and busybox-1.32.0. As we also used the ARM ISA for the first contribution on QEMU scalability, we manually cross-compiled busybear to run in ARM in addition to RISC-V. For our second contribution we stayed only with busybear in RISC-V and we used linux-5.15.32, dropbear-2020.81 and busybox-1.35.0.

Benchmarks

We used two different sets of benchmarks to do the evaluation of our two contributions. The first one is the PolyBench/C [PY15] which is a collection of around thirty programs.

All the programs are single-threaded and are written in C. They address multiple computing domains such as linear algebra, image processing, physics simulation, statistics and much more. The second one is the PARSEC suite for Princeton Application Repository for Shared Memory Computers. They are multi-threaded programs and we gave more comprehensive information about them in Section 4.3.2 as they are an important asset in the evaluation of QEMU scalability.

6.2 Scalability of QEMU Parallel Implementation

In this section we report the measured execution times of the PARSEC suite (natively or simulated). We used it with the LARGE inputs. On each plot, the vertical black line represents the number of host CPUs. The vertical red line is used to represent when the SMT feature is not enabled, meaning only one hardware thread per core is activated. On all the plots, we decided to show the wall-clock times. Another solution could have been to display the speed up but as lots of programs have a similar scalability, it would have been difficult to correctly read the Figures because of many overlaps. Another reason to display the wall-clock time instead of the speed up is that with the wall-clock time we can have information on the order of magnitude of the programs' run-times on QEMU.

Our QEMU scalability evaluation is divided into four sections. In the first section we run natively on the host machine the PARSEC benchmarks. Secondly, we launched QEMU with and without the pinning and we did variations of some configurations that are detailed in the section. Thirdly we used the CPUs isolation and fourthly we analyzed our results. To decide on the number of times to run each benchmark, we manually evaluated the variations and we concluded that 10 is a correct number.

6.2.1 Native execution

Before testing the pinning in QEMU with busybear and with the PARSEC benchmarks, we needed to test the thread affinity proposed by the PARSEC API to see if it has an impact on the execution time of the benchmarks. As explained in Section 4.3.2, we only have access to two environment variables to control the thread affinity. As a result, we cannot decide one by one on each physical CPU each PARSEC thread will run. We can only decide the range of physical CPUs id. All the following Figures in this section are with a log scale on the y axis.

We report in Figure 6.1 and Figure 6.2 the full execution time of the PARSEC benchmarks directly on the Dell PowerEdge host server respectively without thread affinity and with thread affinity. One can see that on both Figures, the full execution time T_{full} decreases as the number of threads increases. Beyond 32 threads, it stays constant.

We compare in Figure 6.3 T_{full} with and without the thread affinity of 4 benchmarks (`ferret`, `bodytrack`, `blackscholes` and `cholesky`). We observe the same behavior for all the benchmarks and Appendix A details the comparison Figures for all other benchmarks. But for the sake of clarity, we decided here to represent only 4 of them.

Cholesky is the fastest benchmark and it executes in less than a second when the number of threads is less than 64. That explains why its behavior is different from the other.

As we can see from the last Figure 6.3, the PARSEC thread affinity does not have an impact on the execution time. It stays the same for most of the programs and even

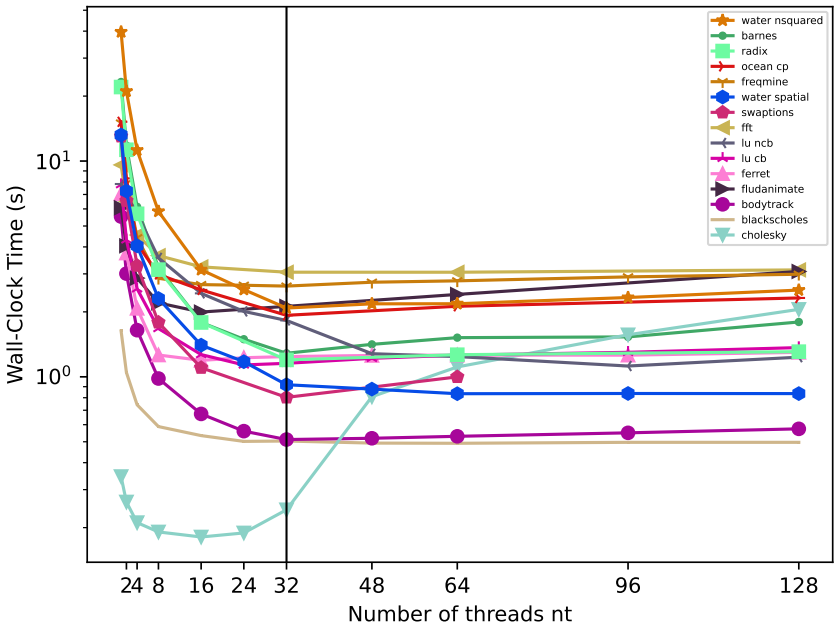


Figure 6.1: Full execution time in **x86 without thread affinity**

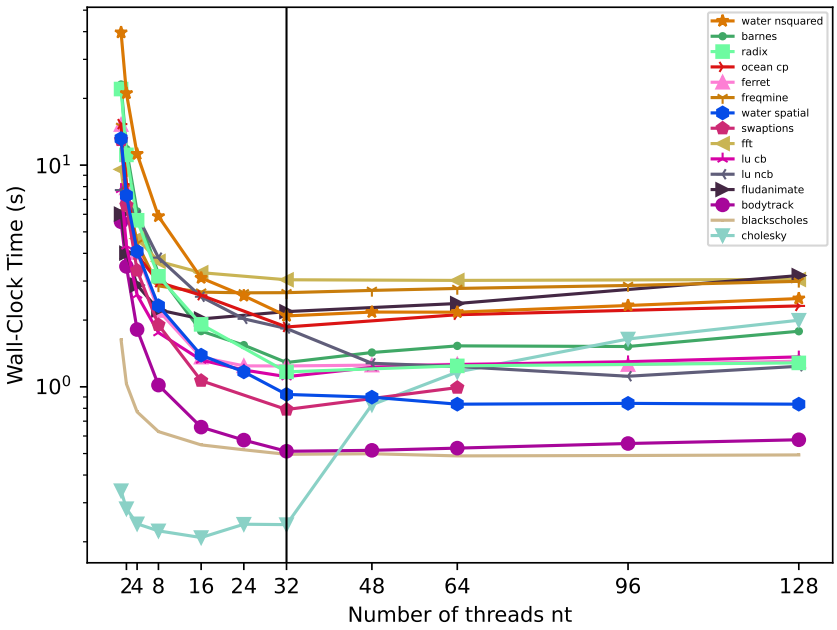


Figure 6.2: Full execution time in **x86 with thread affinity**

degrades the performance a little bit with a small number of threads for a subset of them. Thus, we can conclude that the PARSEC thread affinity does not significantly modify the execution time and that is why we decided to not consider this parameter for the following experiments.

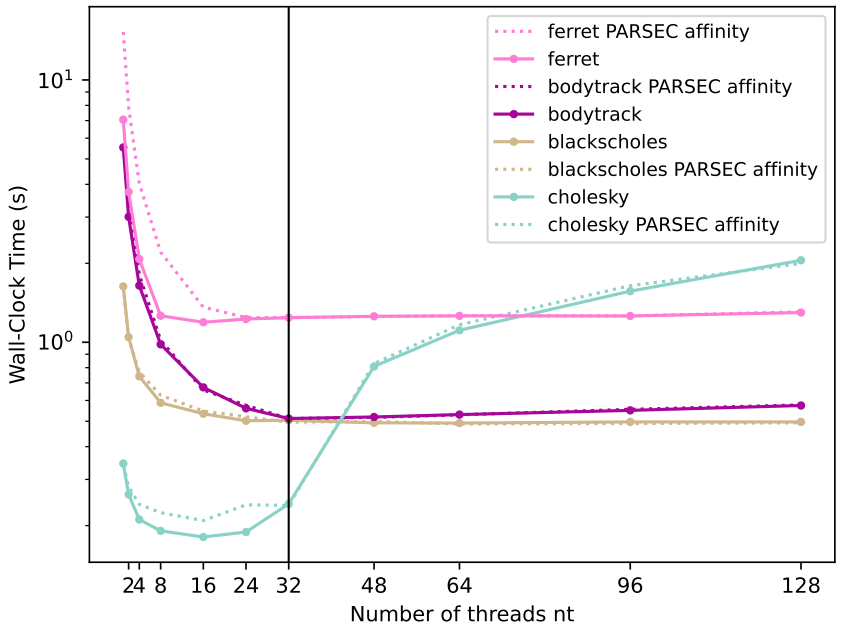


Figure 6.3: Comparison full execution time in **x86**

6.2.2 With/without virtual CPUs pinning

Without pinning

Firstly, we wanted to analyze the scalability of QEMU parallel implementation without the pinning. To do so, we have used different parameters and we have experimented with variations around them.

Figure 6.4 reports T_{full} without the pinning with $n_c = n_t$, meaning that we run QEMU and the PARSEC with the same number of virtual CPUs and threads. For example, when we run QEMU with 4 virtual CPUs, we run all the benchmarks in busybear with 4 threads and we quit QEMU at the end. We do that for $n_c \in \{1, 2, 4, 8, 16, 24, 32, 48, 64, 96, 128\}$. One can see that all the programs take advantage of the parallel execution of QEMU. Until 32 threads (which correspond to the number of host physical CPUs), T_{full} progressively decreases and starts to increase beyond 32. In Figure 6.5, still with $n_c = n_t$, we compare T_{full} with T_{roi} with again only 4 benchmarks for the sake of clarity. Appendix A gives in detail the comparison of T_{full} with T_{roi} for all the programs. T_{roi} corresponds to the execution time of the Region-Of-Interest, *i.e.*, the execution time of only the parallel part of the benchmarks. According to [SR15], the benchmark in the PARSEC that exhibits the better scalability regarding its parallel execution is `blackscholes` as we can see Figure 6.5. For mostly all the programs, T_{full} and T_{roi} are close which means that the parallel part of the programs is an important part of the total execution time. Thus, we can analyze with confidence the scalability of QEMU parallel implementation with T_{full}.

Figure 6.6 compares T_{full} with $n_c = n_t$ and $n_c = 128$. When $n_c = 128$, we run QEMU once with 128 virtual CPUs and we run incrementally the PARSEC benchmarks. The goal of doing this investigation is to determine if inactive virtual CPUs of QEMU can have an impact on the simulation time, *i.e.*, if they induce some overhead. Some benchmarks are clearly more stable than others. For less than 96 threads, T_{full} for

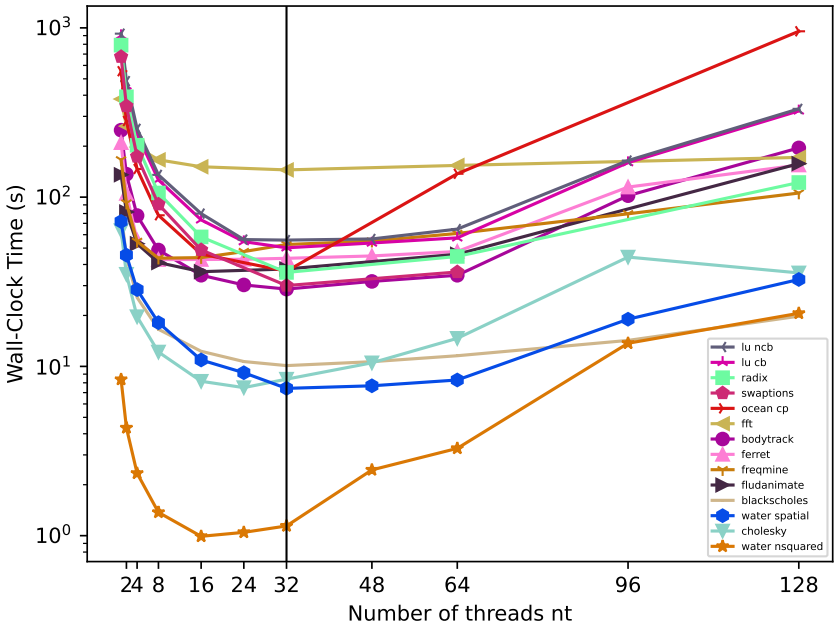


Figure 6.4: Full execution time in QEMU **RISC-V** $n_c = n_t$ **without pinning**

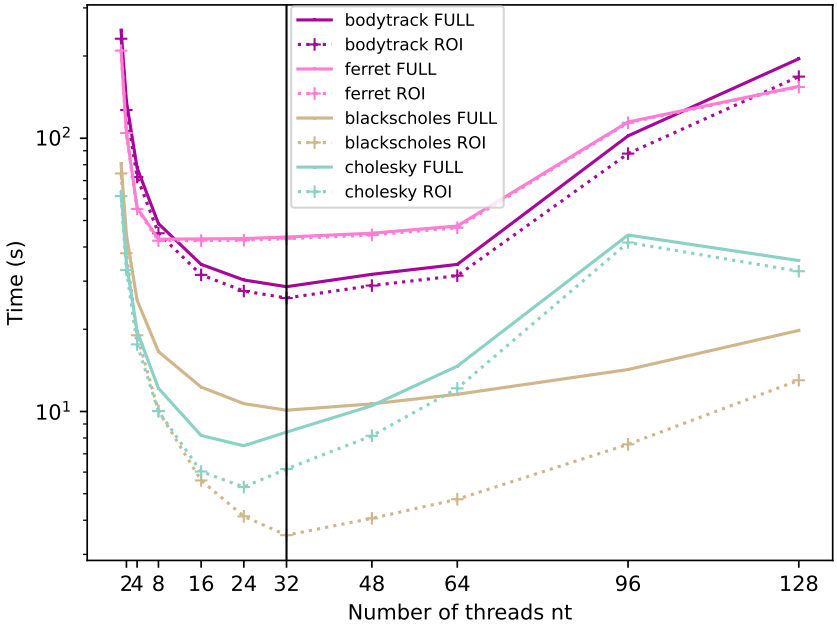


Figure 6.5: Comparison **full** and **ROI** execution time in QEMU **RISC-V** $n_c = n_t$ **without pinning**

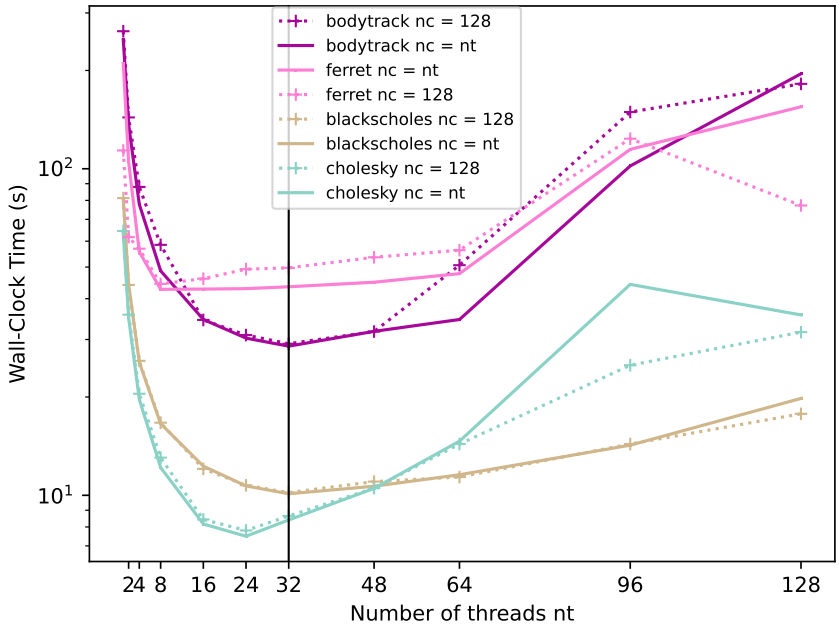


Figure 6.6: Comparison full execution time in QEMU **RISC-V without pinning** with $n_c = n_t$ and $n_c = 128$

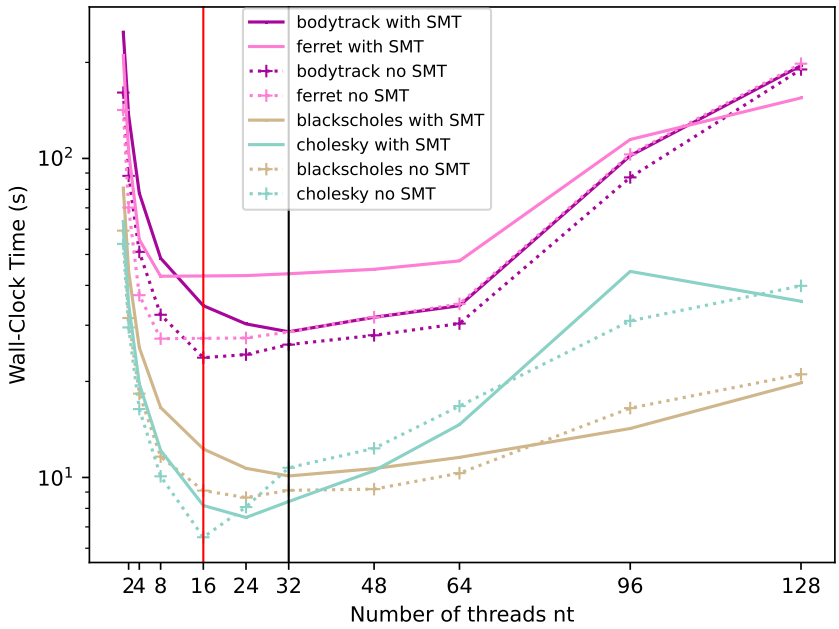


Figure 6.7: Comparison of full execution time in QEMU **RISC-V without pinning** with $n_c = n_t$ for the host machine with and without **SMT**

$n_c = n_t$ and $n_c = 128$ is really similar but for some programs, $n_c = 128$ is a little bit higher than $n_c = n_t$. But overall, there is not a significant difference of T_{full} so we can conclude that inactive virtual CPUs of QEMU do not degrade the performance of the simulation.

In Figure 6.7, we compare T_{full} with and without the SMT configuration. When SMT is enabled, the host machine provides 32 hardware threads, meaning two harts per core (black vertical line). When SMT is disabled, the host machine provides 16 cores with one hardware thread per core (red vertical line). One can see on the Figure that between 1 and 16 threads, the programs are clearly faster when SMT is disabled. The maximum gain in average is 28% at 16 threads. However, beyond 16 threads, the two curves tend to get closer to each other.

With all these experiments and given the shape of all the curves, we can thus conclude that QEMU with the RISC-V target scales well.

With pinning

Now that we have evicted the PARSEC thread affinity and analyzed the scalability of QEMU parallel implementation with the RISC-V target, we can analyze the scalability with the pinning. As for the without pinning analysis, we have used the same parameters and we have experimented with variations around them. Similarly to above, Figure 6.8 reports T_{full} for $n_c = n_t$ and Figure 6.9 compares T_{full} and T_{roi}. Here again, we can see that all the programs take advantage of the parallel execution of QEMU. Appendix A gives in detail the comparison of T_{full} with T_{roi} for all the programs with the pinning.

Figure 6.10 compares T_{full} for $n_c = n_t$ and $n_c = 128$. Here again there is no big differences between $n_c = n_t$ and $n_c = 128$ so we can conclude that inactive virtual CPUs of QEMU do not have an impact on the simulation. Regarding the SMT parameter, Figure 6.11 compares T_{full} with and without the SMT configuration. Like previously without pinning, the curves without the SMT configuration show better performance between 1 and 16 threads. After that, the curves tend to get closer.

We can draw the same conclusion than previously: the parallel implementation of QEMU has a good scalability with the pinning.

Comparison

With all the previous experiments done, we can now compare T_{full} with and without pinning Figure 6.12. Appendix A gives in detail the comparison of T_{full} with and without pinning for all the benchmarks. Cholesky is the only one among the set of benchmarks that is not stable when n_t is high. For all the other benchmarks, the two curves without and with pinning are really close. Surprisingly, the execution time remains the same without and with pinning. Consequently, pinning virtual CPUs in QEMU does not seem to have a positive impact on the simulation performance.

6.2.3 Isolating physical CPUs for QEMU virtual CPUs threads

We wanted to do a supplement analysis to investigate the possible impact of the threads kernel on QEMU running on the host machine. To do so, we isolated physical CPUs for QEMU virtual CPUs threads by using the `isolcpus` Linux kernel configuration. Regarding Figure 4.5, we isolated the physical Cores 1-15 (PU 1-15 and 17-31). Thus, the kernel threads are only able to run on Core 0 (PU 0 and 16). This configuration brings

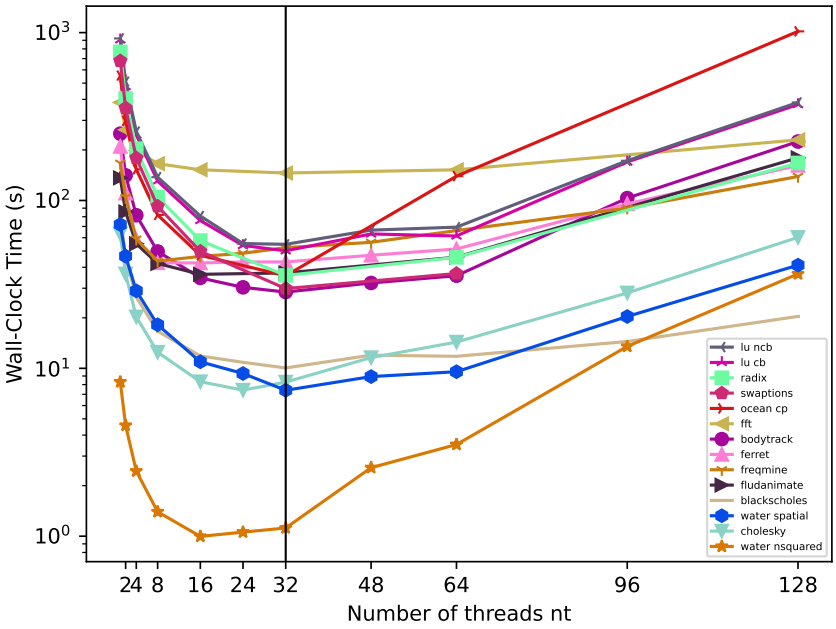


Figure 6.8: Full execution time in QEMU **RISC-V** $n_c = n_t$ **with pinning**

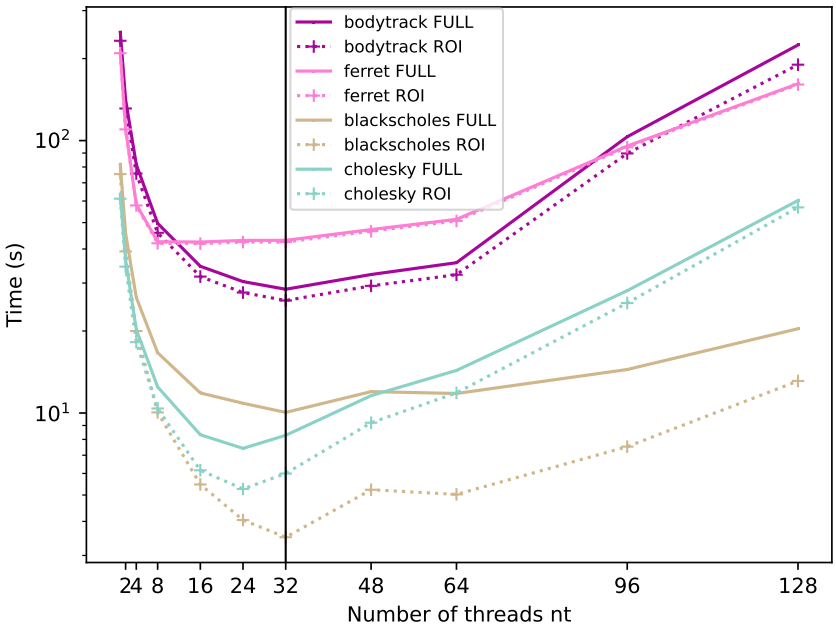


Figure 6.9: Comparison of **full** and **ROI** execution time in QEMU **RISC-V** $n_c = n_t$ **with pinning**

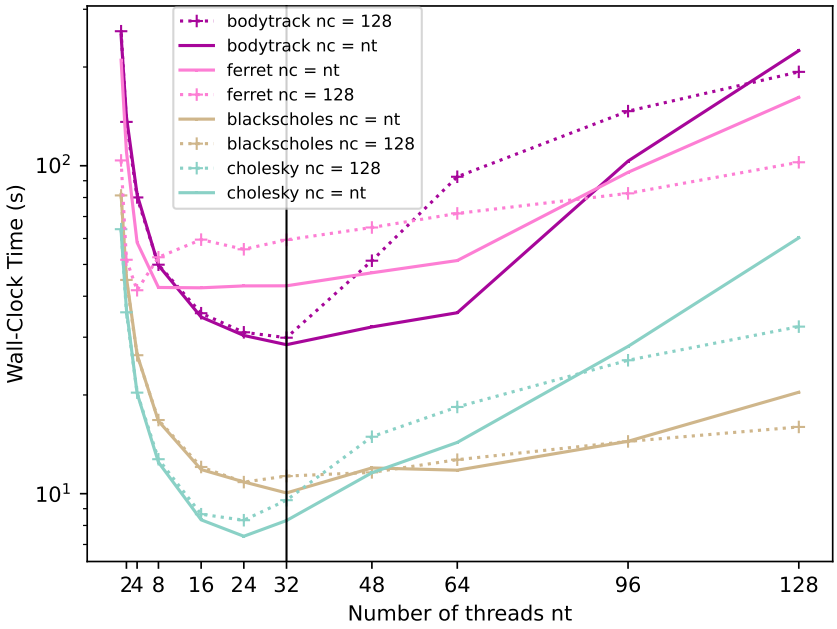


Figure 6.10: Comparison of full execution time in QEMU **RISC-V with pinning** with $n_c = n_t$ and $n_c = 128$

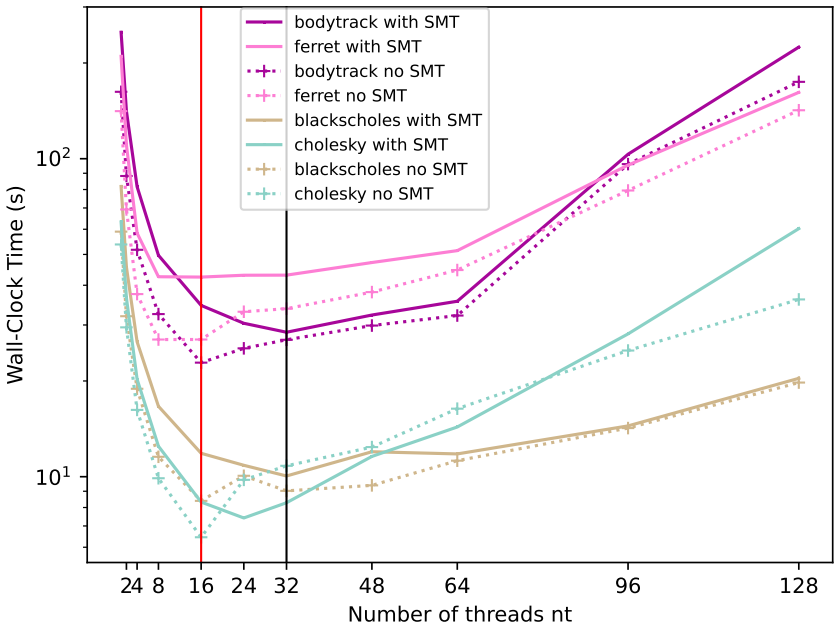


Figure 6.11: Comparison of full execution time in QEMU **RISC-V with pinning** with $n_c = n_t$ for the host machine with and without **SMT**

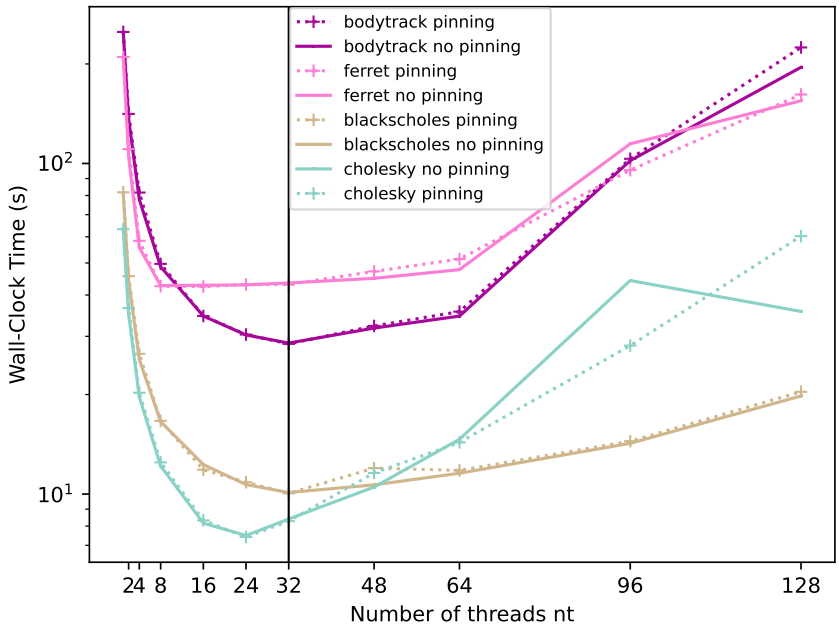


Figure 6.12: Comparison of full execution time in QEMU **RISC-V** $n_c = n_t$ **without pinning** and **with pinning**

two advantages: the first one is that it will mitigate a lot the bottlenecks and the second one is that it will prevent competing between the kernel and QEMU virtual CPUs threads.

Having the isolation of the QEMU virtual CPUs on Cores 1-15 limits the number of threads for the PARSEC and it complicates the ones that require a number of threads equal to a power of 2 to run (which are `fft`, `fluidanimate`, `ocean_cp`, `radix` and `swaptions`).

The comparison with the not pinned configuration is divided into three parts:

- from 1 to 15 threads: no differences for all the benchmarks except one. We have an improvement of 7% for `lu_cb` with 15 threads compared to the not pinned configuration.
- from 60 threads and more: some differences, improvements as degradations but mostly degradations.
- for 30 threads: this is where we have the most improvements for 4 benchmarks. We have a range of improvement between 4% and 16%. We have a degradation for the other benchmarks.

The improvements of the last part for 30 threads are represented Figure 6.13. The pinned and not pinned configurations were done with 32 threads. For the isolcpus configuration, we were able to run the programs with 30 threads as Core 0 is for the kernel threads.

One can see that we have some improvement of the execution time for `cholesky`, `freqmine`, `lu_cb` and `lu_ncb`. For the other benchmarks, it either degrades or does not impact the execution time. In conclusion, combining pinning and CPUs isolation can significantly reduce the execution time in some cases but it is not generalizable for all

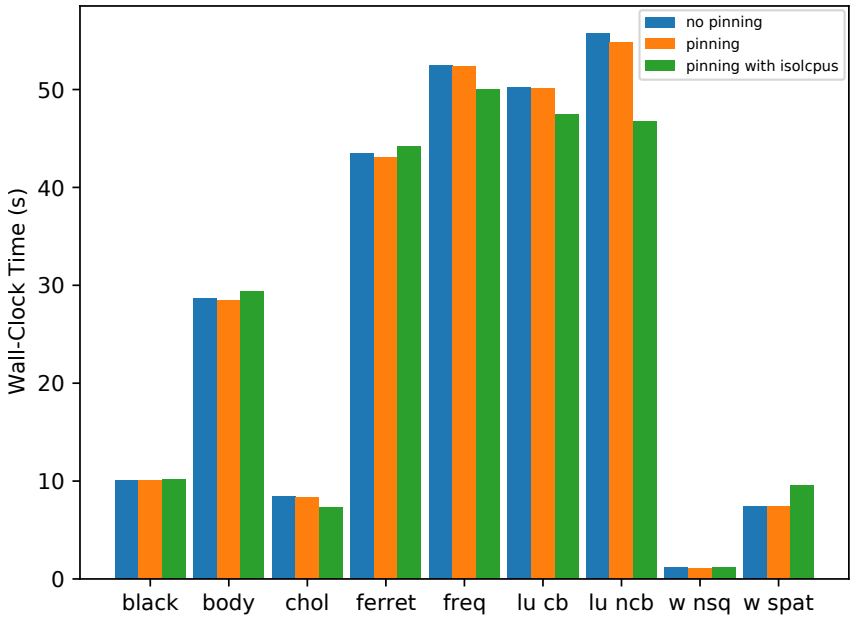


Figure 6.13: Comparison of the full execution times in QEMU **RISC-V** for 32 threads without pinning, with pinning and with isolcpus pinning

the benchmarks and for all numbers of PARSEC threads.

6.2.4 Is pinning helpful?

Remark for a large number of QEMU virtual CPUs

When we compare the curves of the full execution time of the native execution in x86 and the simulated one in QEMU RISC-V, the main difference lies when the number of PARSEC threads goes above 32 (which correspond to the number of physical CPUs on the host machine). For the simulated one, the curves increase sharply. To explain this behavior, we used the Linux `perf` tool [DM10, Gre13] to retrieve the number of CPU cycles during the execution of one benchmark in busybear running in QEMU. We chose `lu_cb` as it exhibits the better properties that we want to analyze and we recompiled QEMU to disable the optimizations. We focused on the ROI section of this benchmark. The investigation showed us that approximately 70% of the time was spent in the Linux `futex` syscall used by the global mutex of QEMU. When the simulation is done with a lot of virtual CPUs, a bottleneck happens since the code generator's interrupt handling function requires a lock on this mutex.

Using ARM instead of RISC-V as the QEMU target

With all the experiments done with the RISC-V target in QEMU, we concluded previously that the pinning does not have a positive impact on the execution time. To make sure that this behavior is not target dependent, we have done the same experiments using ARM as the target. We focused on when $n_c = n_t$ and without the PARSEC thread affinity. Because the machine `virt` limits the number of cores with ARM in QEMU, it was only possible to experiment until 96 virtual CPUs.

We report Figure 6.14 and Figure 6.15 T_{full} for all the benchmarks with QEMU ARM respectively without and with the pinning configuration. We have the same behavior

	CPU migrations		L1-dcache-load-misses (10^6)	
n_t	no pinning	pinning	no pinning	pinning
1	1 014	1 155	79 469	81 340
2	884	799	82 478	97 332
4	2 706	705	86 350	102 624
8	7 930	84 687	95 995	114 283
16	23 692	91 564	114 800	135 526
24	29 404	96 639	60 700	64 071
32	1 965 662	1 345 667	168 547	164 196
48	8 411 897	260 084	96 999	91 701
64	17 141 497	722 055	258 957	254 168
96	62 427 446	689 736	500 362	465 388
128	324 423 980	2 958 824	2 089 470	2 347 334

Table 6.1: Perf for QEMU RISC-V without and with vCPUs pinning

as in QEMU RISC-V: between 1 and 32 threads, the curves decrease progressively and after 32 threads it goes up. The QEMU scalability is still good with the ARM target.

Figure 6.16 reports the comparison of T_{full} without and with pinning for 4 benchmarks in QEMU ARM. We obtain the same observation for all the other benchmarks. Appendix A gives in detail the comparison of T_{full} with and without pinning for all the benchmarks. Similarly than with RISC-V, the pinning does not improve the execution time.

Pinning investigation with Perf

To understand why the pinning does not improve the performance, we have done a complete investigation with `Perf`. Here again we focused on $n_c = n_t$. The following tables show the result of the record of Perf on all the benchmarks run 10 times in RISC-V and in ARM (respectively Table 6.1 and Table 6.2).

	CPU migrations		L1-dcache-load-misses (10^6)	
n_t	no pinning	pinning	no pinning	pinning
1	253	300	254 493	263 479
2	630	112	251 762	321 809
4	2 131	76	282 065	371 609
8	8 579	726	329 442	417 983
16	84 640	438 471	450 250	565 866
24	341 051	33 784	495 140	508 461
32	26 675 217	5 189 901	1 046 410	975 366
48	354 436 238	3 027 408	1 165 892	913 236
64	744 645 854	7 170 919	1 911 263	1 589 535
96	919 101 918	4 913 936	1 809 297	1 454 464

Table 6.2: Perf for QEMU ARM without and with vCPUs pinning

Both tables show the evolution of the number of thread migrations and L1 cache misses in QEMU with and without virtual CPUs pinning. Migrations happen even with the pinning because some tasks such as peripherals are threaded in QEMU. In RISC-V

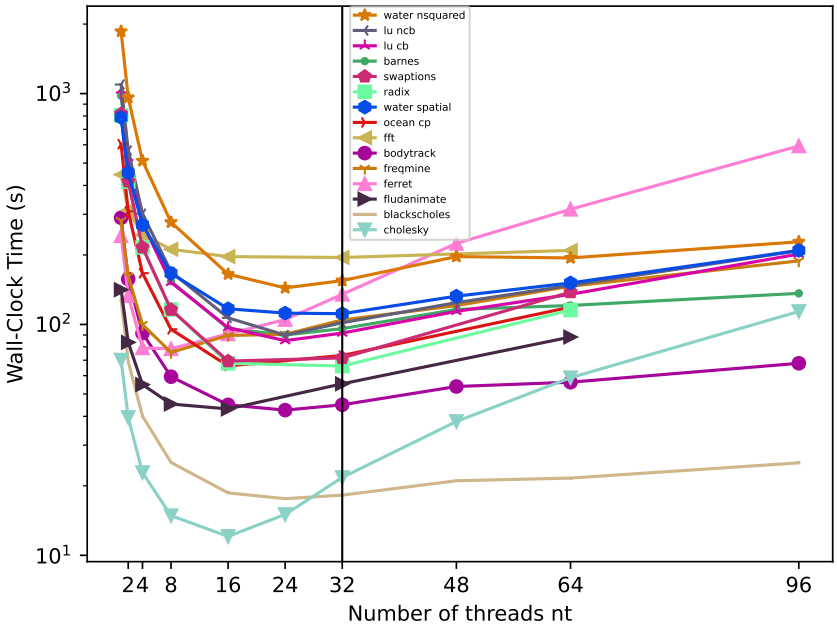


Figure 6.14: Full execution time in QEMU **ARM** $n_c = n_t$ **without pinning**

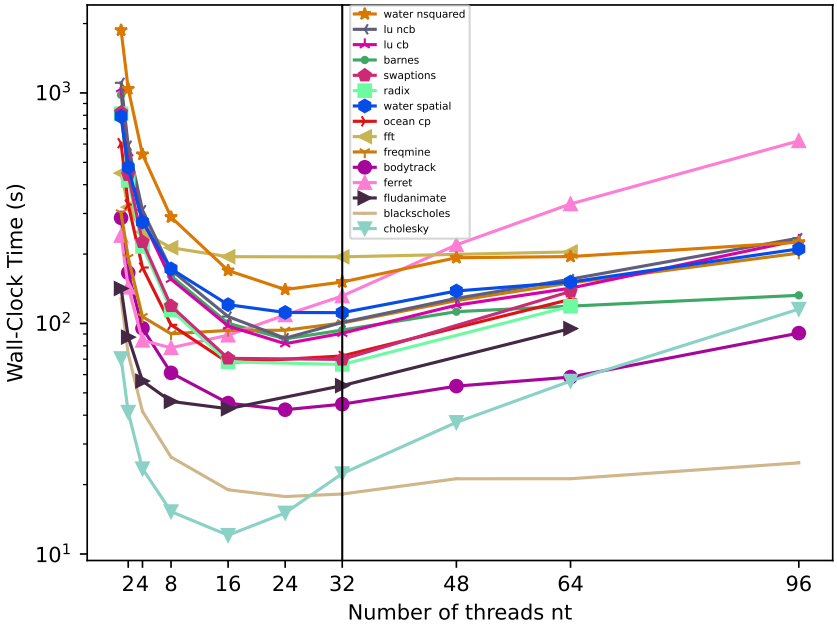


Figure 6.15: Full execution time in QEMU **ARM** $n_c = n_t$ **with pinning**

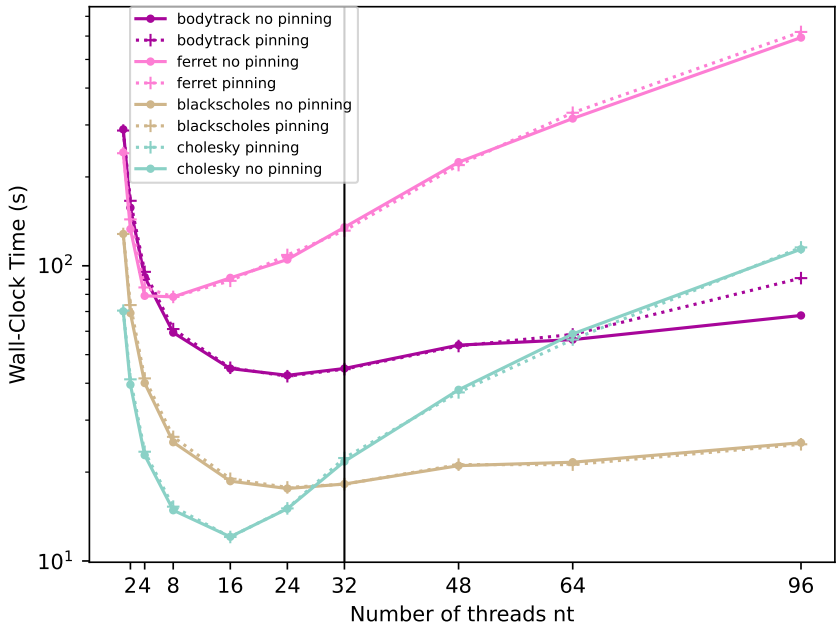


Figure 6.16: Comparison of full execution time in QEMU **ARM without pinning** and **with pinning**

and in ARM, one can see that the number of CPU migrations increases as the number of threads n_t increases. We remember that not all the benchmarks can be run with n_t that is a power of 2. When n_t is not a power of 2, only 9 benchmarks can be run so that is why on the tables the values of the metrics can be lower than others. Except for some value of n_t under 32, overall, there is less CPU migrations with pinning in both QEMU RISC-V and ARM. We have a huge reduction of the migrations with $n_t = 128$ (more than 100 times). However, regarding the L1 cache misses, it stays constant without and with pinning in RISC-V and ARM, which can explain why the execution time does not vary without and with pinning.

Standard deviation

As the last investigation, we computed the standard deviation on all the execution times with and without pinning for both the RISC-V and ARM targets. According to the literature, the pinning seems to be related with the stability of the execution times. The idea is that a lesser disparity in execution times would be an advantage and therefore the measurement of standard deviations will quantify it. However, the execution times are not more stable with pinning in general and depend on the benchmark. Thus, we cannot conclude that the stability is better with the pinning.

Analysis conclusion

Even if the pinning decreases significantly the number of CPU migrations when n_t is high, it is not enough to have an impact on the execution time. The fact that the L1 cache misses are close without and with pinning explains why we have not seen any improvements with pinning. In the end, we conclude that the pinning does not have a positive impact on the execution time and even for some benchmarks, it degrades

the performance a little bit. As stated by [GDBAS20], pinning virtual machines does not ameliorate the performance if using CPU-bound applications. Unfortunately, the PARSEC applications are CPU-bound, as is QEMU, which explains why we do not see any improvements. We cannot beat the current Linux scheduler as it attaches a thread to a physical CPU as long as it does not degrade the performance. Recent works have shown that pinning is only beneficial for a certain type of applications [PBC+15].

6.3 Instruction cache (L1i) evaluation

This section focuses on the evaluation of our solution on the L1i model that we implemented thanks to the TCG Plugins. Here we used PolyBench/C and PARSEC as sets of benchmarks (the first one is single-threaded and the other is multi-threaded). The PolyBench/C set was run with the MEDIUM inputs and the PARSEC set with the LARGE inputs. Our L1i model is not target dependent but as we needed to choose one to run our experiments and because it is widely used in research, we worked with the RISC-V target. Compared to the previous section of QEMU scalability where we used QEMU only in `full-system` mode by running busybear Linux in it, here we also used QEMU in `user`-mode. It is a mode where the host OS directly handles the system calls and signals. As explained in Section 5.2.5, we need to make a fair comparison of speed with the existing cache plugin and because of the dependency on the flow of executed target instructions, we used the `user`-mode. Executing the PARSEC benchmarks in `full-system` with busybear and in `user`-mode are two completely different ways of doing it. Due to some argument constraints and other reasons which have not yet been investigated, we were able to correctly run 10 benchmarks of the PARSEC in `user`-mode, compared to 14 in `full-system`. However, the 10 benchmarks are different enough to cover various properties and therefore be reliable for performing our analysis.

Since we run QEMU in its multi-threaded version, the results are not deterministic. That is why we run our experiments 20 times that, given the standard deviation we computed, gives us confidence in the results obtained. For all the Figures in this section, we use `cache` to refer to the cache plugin that is shipped with QEMU, `cacheTB` to refer to the plugin implementing our solution and `vanilla` to refer to QEMU without any plugin activated.

All the following experiments are done with this arbitrary instruction cache configuration: 8-way, 32-set, 64 bytes per line and LRU. Since we model only the L1i, the notion of Write Through and Write Back Allocate does not matter. Moreover, there is no prefetching of the cache lines.

6.3.1 Issues mitigation

Before doing the statistical validation of our L1i model and the comparison with the existing cache plugin, we must come back first to the issues that we encountered and see how we mitigate them.

Error due to exceptions in TBs

As seen in Section 5.2.4, early TBs exits can happen. Since it impacts the number of instructions counted in each TB, we need to make sure that the consequence of this

issue is negligible on our cache statistics. To do so, we decided to analyze two experiments: the boot of busybear Linux and the execution of the LU decomposition of a 2048×2048 matrix. We run these experiments with a single virtual CPU. We recorded 4 statistics: the number of TBs executed, the number of early TBs exits, the number of instructions executed and the number of instructions wrongly counted (meaning that they are counted by our model while they should not). In average, the length of a TB is 5.51 target instructions for the boot of busybear Linux and 16.61 target instructions for the LU decomposition program. Table 6.3 sums up the results for the two experiments. We can clearly see that the number of wrongly counted instructions is very negligible relatively to the total number of instructions counted. The effect on our cache statistics is thus not visible.

	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

Table 6.3: Measure of the error due to early TB exits.

We still decided to implement a small correction in QEMU by forcing a TB to end at each memory access (thus avoiding the page-fault issue). But as a result, the number of TBs executed for the boot of busybear was multiplied by 4 and the simulation time increases by 50%. In the end, it is a predictable behavior as 44% of the instructions are memory accesses in this case. Even if we can correct this issue, it is not useful to mitigate it as it costs an important overhead while not having a visible impact on our cache statistics.

Dependency on Simulator Runtime

When we record the execution time of only the boot of busybear even though there is a dependency on the simulator runtime, we obtain the Figure 6.17. We varied the number of virtual CPUs from 1 to 128 with values that are powers of 2. We have in general a noticeable improvement with our plugin compared to the existing plugin and it is even more noticeable with 64 and 128 virtual CPUs (factor of 3).

However, as the cache statistics are significantly affected by the timer interruption that will generate instructions, we cannot conclude with the `full-system` mode on the performance of our plugin.

We wanted to try to implement a solution to bypass the issue. The TCG Plugins API does not allow us to retrieve a lot of information from QEMU execution. But to do so, we did a trick in the QEMU source code to stop counting ticks during our cache model simulation. By adding extra function pointers in the structure used to share system information with the TCG Plugin, we were able to call inside our plugin the two functions `cpu_enable_ticks` and `cpu_disable_ticks` which respectively enable and disable the clock. Thanks to this tricky solution, we can disable the count of the ticks during execution each time we do our cache simulation on the instructions in a TB. This solution is also applicable for the existing cache plugin.

To see if our solution produces consistent statistics, we recorded the total instructions counted and the execution time of the boot of busybear Linux without and with the enabling/disabling of the ticks' solution on both our `cacheTB` plugin and the existing `cache` plugin. The issue on the simulator runtime dependency also affects the data

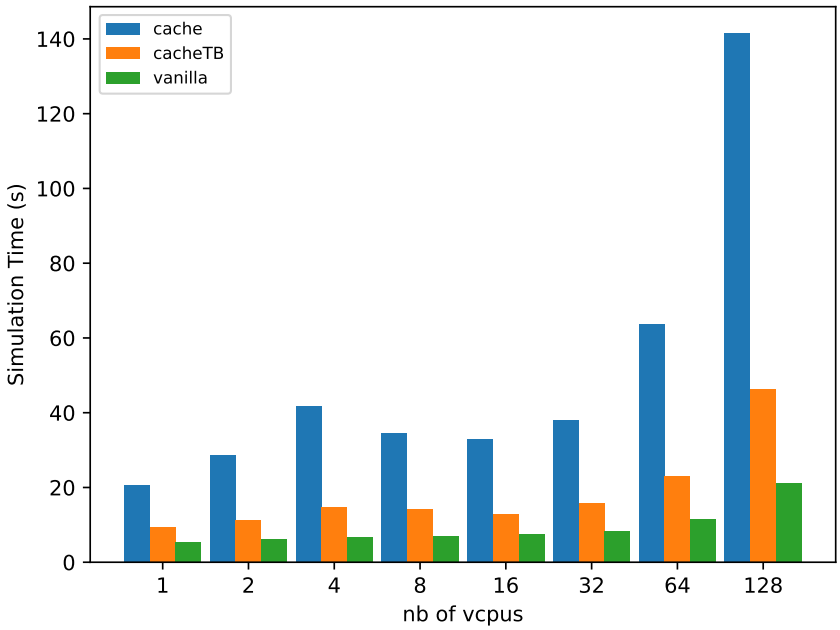


Figure 6.17: Execution time of only the boot of busybear

cache simulation present in the existing cache plugin, so when we implemented the solution, we disabled the count of the ticks not only for the instruction cache but also for the data cache simulation. The following Table 6.4 and Table 6.5 sum up the total of instructions counted and the execution time of only the boot of busybear Linux for 1, 2 and 4 virtual CPUs. Because of the huge impact on the execution time when disabling and enabling the counting of the ticks for each TB, we were only able to execute up to 4 virtual CPUs. There are differences of several million instructions with and without ticks counting which is definitely not negligible.

	cache (with/without)	**cacheTB (with/without)**
1 vCPU	372 674 140 / 384 397 519	361 544 214 / 363 966 530
2 vCPUs	496 654 744 / 536 769 811	495 187 065 / 519 242 982
4 vCPUs	799 097 859 / 3 420 291 895	740 926 188 / 3 053 453 598

Table 6.4: Mean **number of instructions** of only the boot of busybear with and without ticks counting

	cache (with/without)	**cacheTB (with/without)**
1 vCPU	30.56s / 70.07s	20.72s / 38.08s
2 vCPUs	44.49s / 139.56s	27.98s / 64.94s
4 vCPUs	66.33s / 2 684.21s	33.60s / 430.64s

Table 6.5: Mean **execution time** of only the boot of busybear with and without ticks counting

A behavior not represented on the Tables that we observe is that for 1 virtual CPU, when we disable/enable the ticks count, the total number of instructions counted seems to vary less than the classical execution. However, this observation only happens for 1

virtual CPU. For 2 and 4 virtual CPUs the total number of instructions counted varies a lot from one execution to another. In addition to the variations, the execution time is also considerably affected and is very unstable for 4 virtual CPUs as it goes from 370 to 6 527 seconds for the `cache` plugin and from 170 to 1091 seconds for our `cacheTB` plugin on 20 executions.

Several things can be the cause of our not working solution. One main possibility is that we did not take care of the impact this has neither on QEMU forward progress possibilities, nor on other clocks in QEMU. This issue in `full-system` is thus not resolved yet at the time of this writing.

6.3.2 Statistics validation

To make a fair comparison of the performance of our plugin with the existing one, we first need to validate our produced statistics. To do so, we compared the total number of instructions and the number of misses. We used the PolyBench/C suite to do the validation in mono-core and the PARSEC suite for the multi-core validation.

Mono-core: PolyBench/C suite

We run the entire suite of 28 programs (except the one that failed during execution which is `durbin`) in QEMU on "bare-metal". As the behavior is deterministic, it was simple to compare the statistics produced by our plugin `cacheTB` and the existing `cache` plugin. We obtained exactly the same number of instructions and the number of misses so we can validate the good behavior of our cache model in mono-core. To further validate our cache model in the context of DBT simulation, one can compare it with another simulator. Section 6.3.4 deals with this by using the gem5 simulator.

Multi-core: PARSEC suite

Regarding the multi-core validation, we run the 10 benchmarks of the PARSEC and we computed the mean of the total number of instructions and the number of misses. We varied the number of virtual CPUs with the following values: $\{1, 2, 4, 8, 16, 32, 64, 128\}$. We increase at the same time the number of threads of the PARSEC benchmarks with the number of virtual CPUs. Because of the dependency on the runtime when using QEMU in `full-system` with busybear, we run the programs in `user`-mode which means that they run directly in QEMU without the simulation of a Linux system.

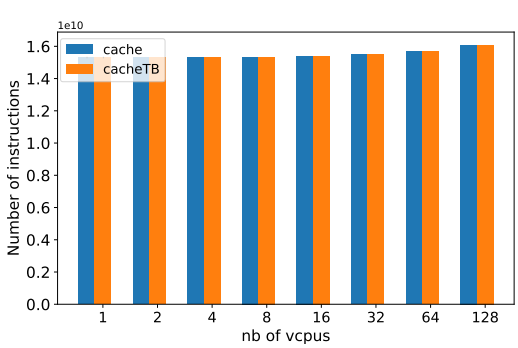


Figure 6.18: Total **number of instructions** for lu_cb (log scale on x-axis)

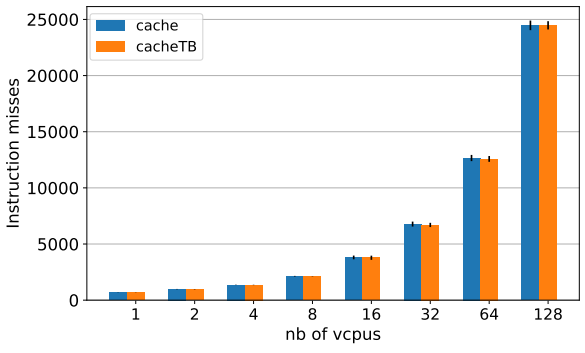


Figure 6.19: Total of **instruction misses** for lu_cb (log scale on x-axis)

Figure 6.18 and Figure 6.19 show respectively the total of instructions executed and the total of instruction misses for the benchmark `lu_cb`. The value for each number of virtual CPUs is the mean. One can also see on the Figures some vertical black lines. As we have run our experiments 20 times, these lines correspond to the minimum and maximum value obtained. It gives us an idea of the range of the values for the 20 executions. These lines are almost not visible on Figure 6.18 and the range is very small on Figure 6.19. Appendix B contains the Figures for the other benchmarks. Except for cholesky whose execution is not very stable (as seen in Section 6.2), we clearly see that the total number of instructions and instruction misses are remarkably similar with our `cacheTB` plugin and the `cache` plugin for all the other benchmarks. Thus, we can conclude on the validity of our statistics in multi-core.

6.3.3 Simulation time

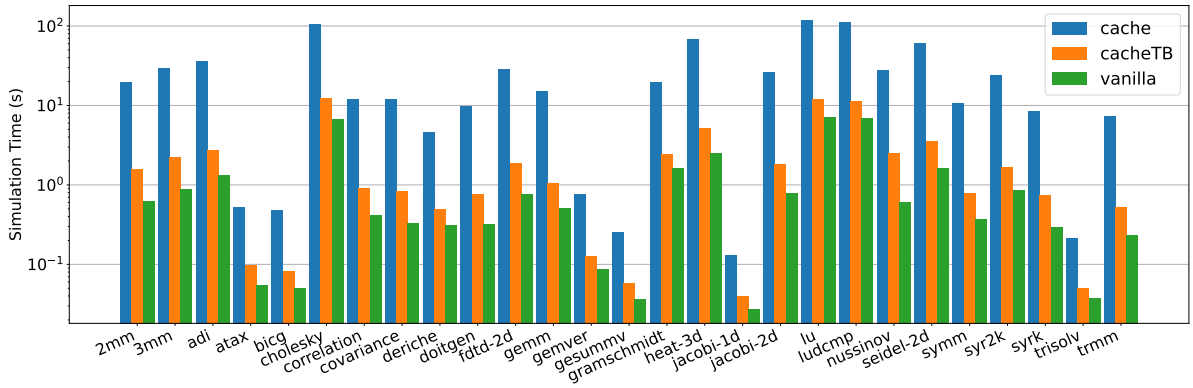


Figure 6.20: Simulation time of the PolyBench/C programs (log scale on y-axis)

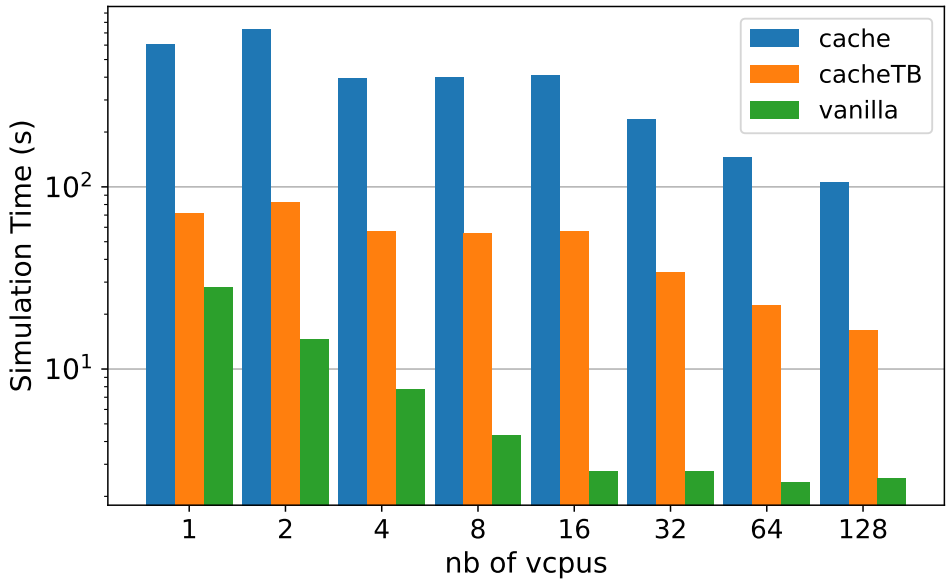


Figure 6.21: Simulation time of lu_cb (log-log scale)

Now that we have validated the good behavior of our cache, we can compare the simulation time of the QEMU `cache` plugin with our `cacheTB` plugin. Figure 6.20 shows

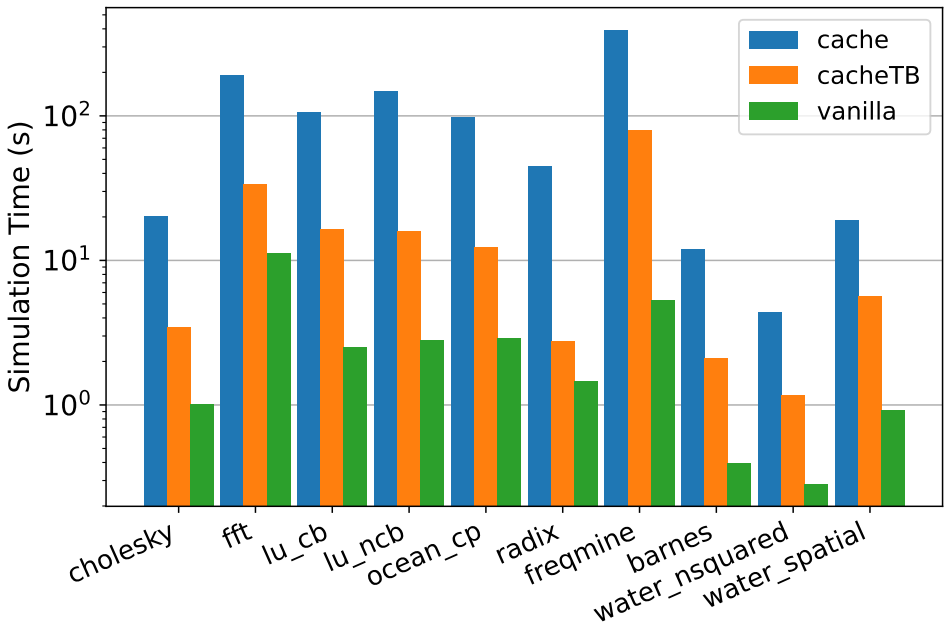


Figure 6.22: Simulation time of the PARSEC programs on 128 vCPUs (log scale on y-axis)

the mean of execution time for each program of the PolyBench/C suite in bare metal in QEMU. The simulation times of the y axis are on a log scale. Our plugin is definitely faster than the existing one for all the programs of the suite. For the multi-core analysis, Figure 6.21 represents the comparison of the simulation time of the PARSEC program `lu_cb` which was run in `user`-mode in QEMU with 1 to 128 virtual CPUs and Figure 6.22 represents the simulation time with 128 virtual CPUs for all the PARSEC programs. Appendix B gives in detail the histograms of the simulation time for all the PARSEC programs as we have the same behavior. Here again in multi-core we clearly see that our plugin is considerably faster than the existing plugin.

	PolyBench/C	PARSEC
Speedup of **cacheTB** over **cache**	10.87	7.18
Slowdown of **cache** over **vanilla**	23.67	59.85
Slowdown of **cacheTB** over **vanilla**	2.07	10.16

Table 6.6: Mean simulation time ratios.

We summarize Table 6.6 all the different mean speedup/slowdown ratios. With the PolyBench/C and the PARSEC suites, our `cacheTB` plugin is 7 to 10 times faster than the existing `cache` plugin. When comparing with QEMU `vanilla`, *i.e.*, QEMU without any plugin, our plugin degrades the performance less than the existing one (only a maximum of 10 times slower, which is already quite a lot, compared to 60 times slower with the `cache` plugin).

6.3.4 gem5 comparison

Doing the statistical validation of our L1i model against the existing cache plugin has allowed us to conclude on the good behavior inside QEMU. However, extending the statistics validation against a completely different cache simulator will bring more con-

tribution to our work. The cache simulator chosen to do the evaluation against is the one available in the gem5 simulator [BBB+11]. It does a more detailed simulation and can simulate in multi-core different cache coherency protocols. The drawback is that the multi-core simulation is done sequentially on one thread and is terribly slow compared to the functional simulation of QEMU. Since it is done on one thread, the simulation is deterministic so only one execution is useful, there is no need to run the simulation 20 times like we did with QEMU. Like QEMU, gem5 also has 2 execution modes: `user` and `full-system` mode. To make objective comparisons, gem5 was executed in `user`-mode and the PARSEC benchmarks used were the same that we ran within QEMU in RISC-V as well as for the cache configuration.

In a multi-core execution, we can force the CPU 0 to always be the one that will launch the system and does all the necessary initializations. The other CPUs are generally blocked on a barrier, waiting for CPU 0 to release it. As a result, in the cache statistics, the CPU 0 will always execute more instructions than the others. The more a multi-threaded program has a sequential part, the more the CPU 0 alone executes instructions. That is why we compared in gem5 and QEMU the number of instructions counted by the CPU 0 and the mean of instructions counted by all the other CPUs.

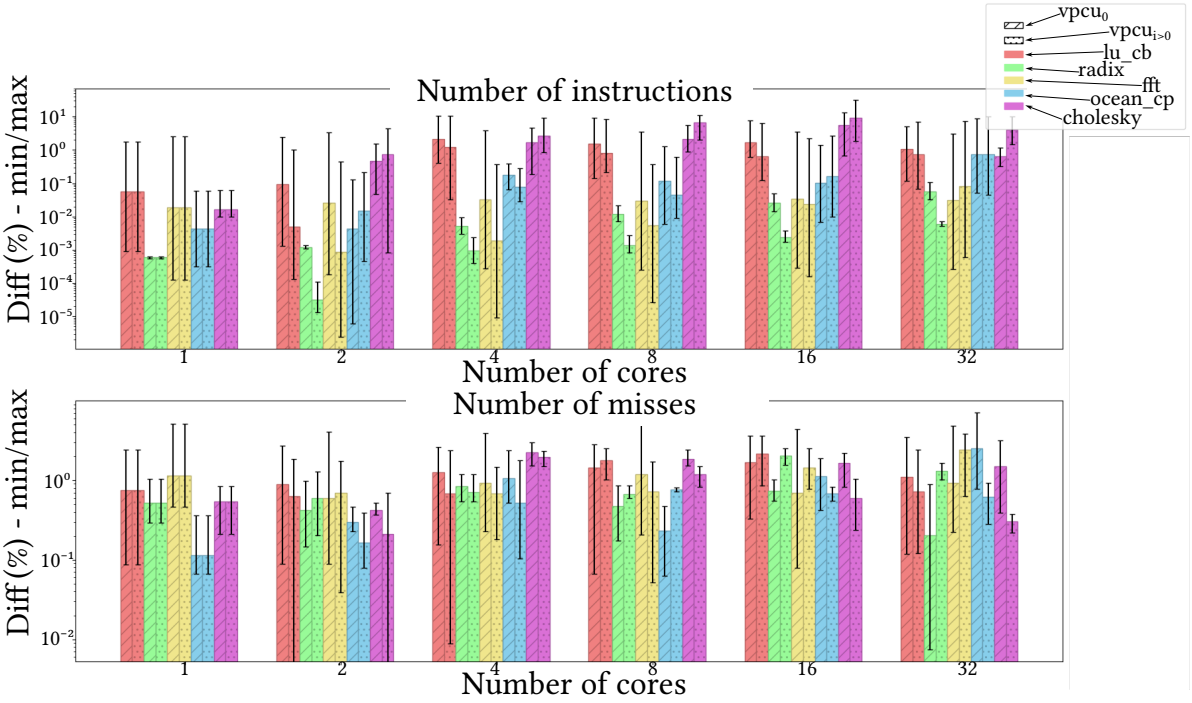


Figure 6.23: Percentage of difference gem5 vs our model in QEMU for the L1i simulation

Figure 6.23 represents the percentage of difference on the total number of instructions and instruction misses in gem5 and our model in QEMU for 5 PARSEC benchmarks: `lu_cb`, `radix`, `fft`, `ocean_cp` and `cholesky`. Due to the slow execution time of gem5 with a high number of virtual CPUs, we limited the simulation to 32 virtual CPUs. For each benchmark in a distinct color, we distinguish the difference on the statistics of only the CPU 0 and on all the other CPUs. Even if the difference seems acceptable (around 0.1% with big variations for the total number of instructions and between 0.1% and 1% for the number of misses), a non-negligible difference of statistics counted in gem5 and QEMU is noticeable. Some reasons behind it were discovered. While investigating the execution trace, we have found that the differences are mainly explained

by execution environment functions and synchronization instructions between the virtual CPUs. In addition, we also discovered another problem: RISC-V instructions can be misaligned, resulting in instructions that can sit on two cache lines and thus can generate 2 cache misses. This behavior is not detected by QEMU. However, due to the nature of the PARSEC which are algorithms with lots of loops, this behavior does not have an impact on the percentage of difference and is not visible.

We carried out the analysis of the differences only on the instruction cache. Differences also happen with the L1d but we did not investigate further. Doing a complete statistics validation against gem5 of an entire cache simulation in QEMU of a L1i, L1d and L2 per core is still at the time of this thesis an ongoing work.

6.4 Data cache (L1d) evaluation

In this section we analyze our data cache L1d model that relies on a separate thread to run the data cache simulation presented Section 5.3.1. To stay consistent with the issue related to the timer in QEMU `full-system`, we have done the evaluation again using the 10 PARSEC benchmarks in QEMU `user`-mode. To communicate with the thread that executes the cache simulation, our L1d model uses a global list of buffers. As a result, before doing the validation of the produced statistics and the comparison of the simulation time, we need to find the optimal value for the size of the buffer and for the number of buffers. Each experiment was done 20 times. In all the Figures in this section, we use `cache` to refer to the existing cache plugin in QEMU, `dcachethread` to refer to the plugin implementing our solution and `vanilla` to refer to QEMU without any plugin activated.

All the following experiments are done with this arbitrary data cache configuration: 8-way, 32-set, 64 bytes per line, LRU and Write Back Allocate.

6.4.1 Optimal buffer size and buffer count

Before doing the statistical validation of our L1d model and the execution time comparison with the existing cache plugin, we must decide on the optimal number of buffer and the optimal size of each buffer. As the principle of our L1d model relies on a separate thread that communicates with our plugin through a list of buffers, a bottleneck may happen. However, the bottleneck is not the same in mono-core and multi-core. In mono-core, only one thread representing the only virtual CPU of QEMU will be interacting with our data cache thread. A bottleneck will happen if there is not enough buffer available and it can degrade the simulation time. Regarding the multi-core execution, we have multiple threads each corresponding to a virtual CPU that interact with our data cache thread. If the number of virtual CPUs is high, here again a bottleneck will happen. To analyze these possible behaviors, we first did an analysis in mono-core with the PolyBench/C suite and then in multi-core with the PARSEC suite.

Mono-core: PolyBench/C suite

We did a complete analysis in mono-core by executing our plugin with the number of buffers that varies in the following values $\{4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096\}$ and the size of each buffer that varies in the same values. We retrieve the execution for

each combination of values. We focused first on 3 benchmarks: `atax`, `doitgen` and `trmm` which have execution times that range from less than a second to less than a minute.

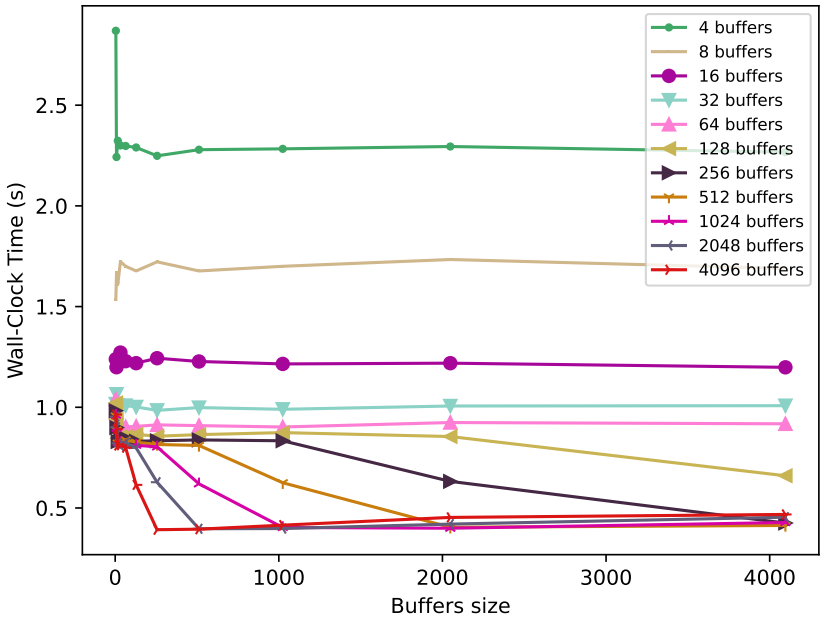


Figure 6.24: Simulation time of `atax` with variations of number of buffers and buffer size

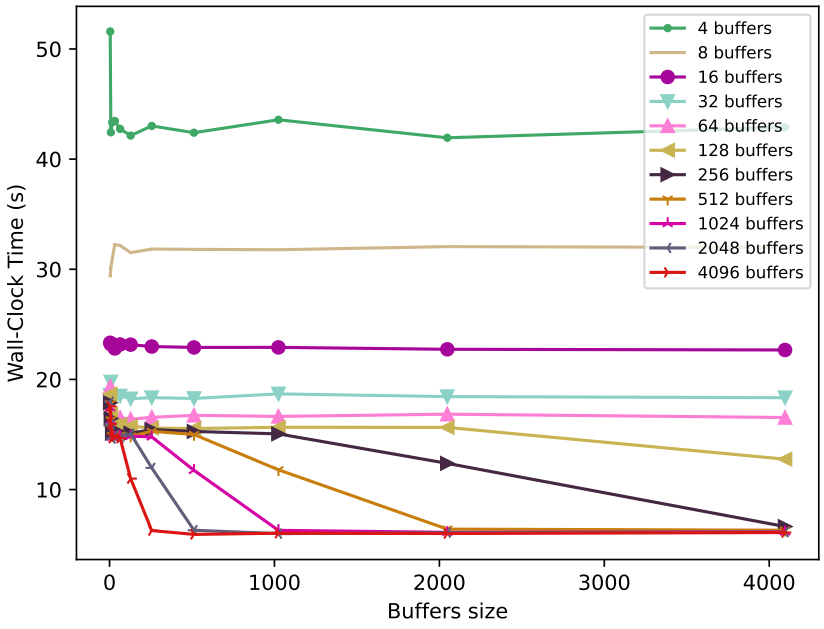


Figure 6.25: Simulation time of `doitgen` with variations of number of buffers and buffer size

Figures 6.24, 6.25 and 6.26 respectively show the evolution of the execution time of `atax`, `doitgen` and `trmm` with multiple variations of the number of buffers and the size of each buffer. One can see that for each number of buffers, the execution time no longer evolves from a certain buffer size. Moreover, with 4, 8 and 16 buffers the execution time is considerably affected. More interestingly, for a number of buffers higher than 256, each of the 3 benchmarks converge to a limit when increasing the size

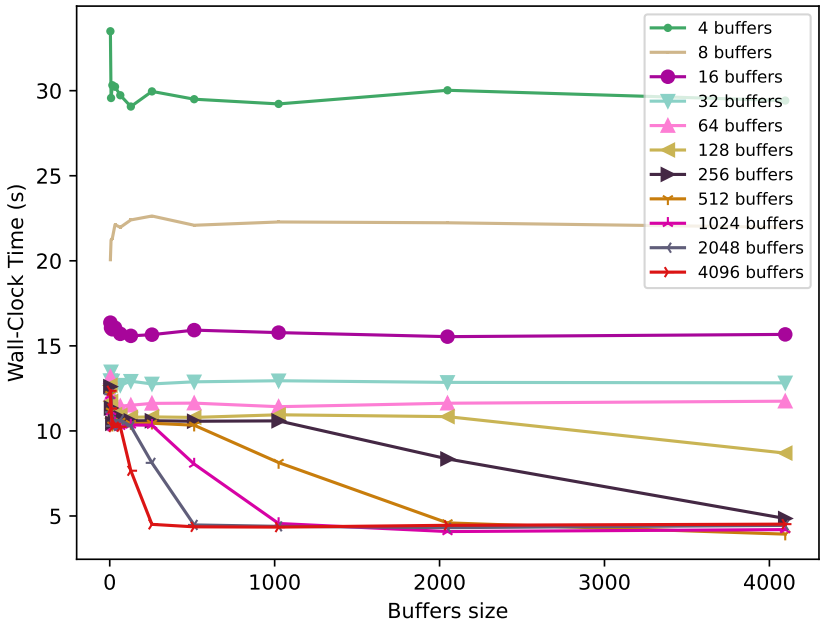


Figure 6.26: Simulation time of `trmm` with variations of number of buffers and buffer size

of the buffers. Thus, at this point, it is no longer interesting to increase the size and the number of buffers. This behavior is not specific to only these 3 benchmarks, we observed the same phenomenon for the other benchmark of the PolyBench/C suite.

To compare the execution time with the existing cache plugin, we must decide a set of values for the size and number of buffers. According to what we can observe on the Figures, we can conclude that 1024x1024 seems to be a viable choice.

Multi-core: PARSEC suite

For the analysis in multi-core, we stayed in `user`-mode with the PARSEC benchmarks. We focused on one arbitrary benchmark of the PARSEC suite: `water_nsquared`. Indeed, testing all the benchmarks costs a huge amount of time (execution time x number of benchmarks x number of buffer configurations x 20). That is why we chose `water_nsquared` as it executes in a correct amount of time and also to differ from the classical `lu_cb` benchmark that we tend to choose. We executed our L1d model in multi-core with the number of buffers that varies in the following values $\{256, 512, 1024, 2048, 4096\}$ and the size of each buffer that varies in the same values. Analysis with a small number of buffers and a small size of each buffer is useless as we will have for sure bad performances. Regarding the number of virtual CPUs, we tested the following values $\{1, 2, 4, 8, 16, 32, 64\}$.

Finding an optimal combination of number of buffers and buffer size in multi-core is a more complex task. An optimal combination of values for a given number of virtual CPUs might not be the same optimal for another number of virtual CPUs. Moreover, it is likely to be benchmark dependent because of the scalability. That is why we focus here on the analysis of one benchmark.

Figure 6.27, Figure 6.28 and Figure 6.29 respectively illustrate the evolution of the simulation time with variations of the size and number of buffers for 4, 8 and 64 virtual CPUs. One can notice that the optimal combination of values seems to be 4096 buffers of a size 256 for 4, 8 and 64 virtual CPUs and even if it is not represented here, it is also

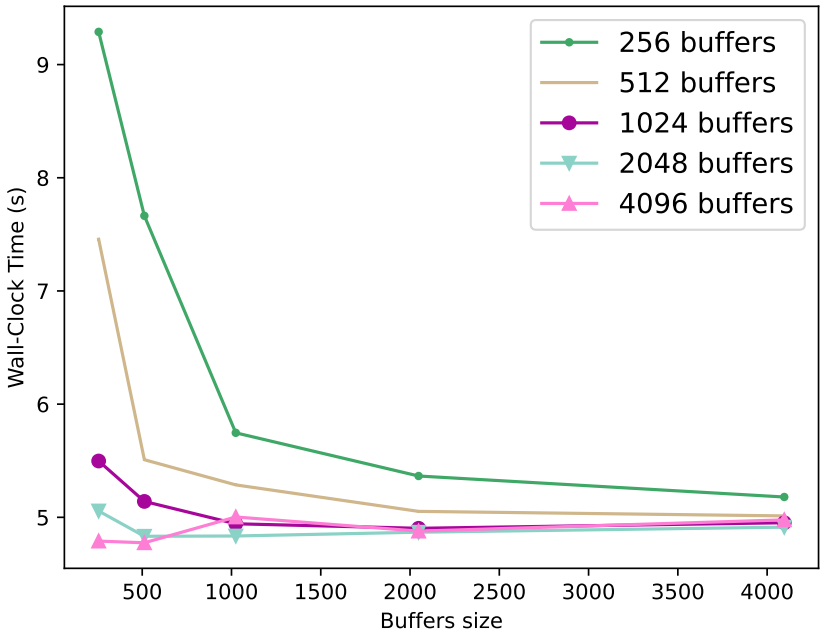


Figure 6.27: Simulation time of the PARSEC `water_nsquared` with variations of number of buffers and buffer size with **4** vCPUs

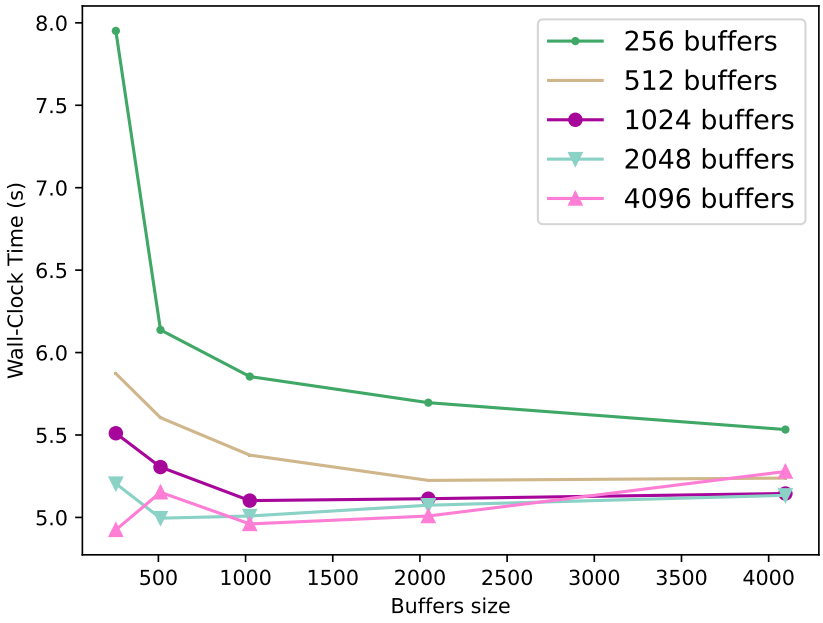


Figure 6.28: Simulation time of the PARSEC `water_nsquared` with variations of number of buffers and buffer size with **8** vCPUs

the optimal values for 1, 2, 16 and 32 virtual CPUs. Appendix C presents the Figures for 1, 2, 16 and 32 virtual CPUs. With this observation, a global remark can be made: in order to achieve the best performance when increasing the number of virtual CPUs, we need to decrease the size of each buffer and increase the number of buffers. When the number of virtual CPUs is high (like with 64 on Figure 6.29), increasing the size of the buffers has a negative impact on the performance.

Thus, the best configuration for our L1d model in multi-core seems to be a high

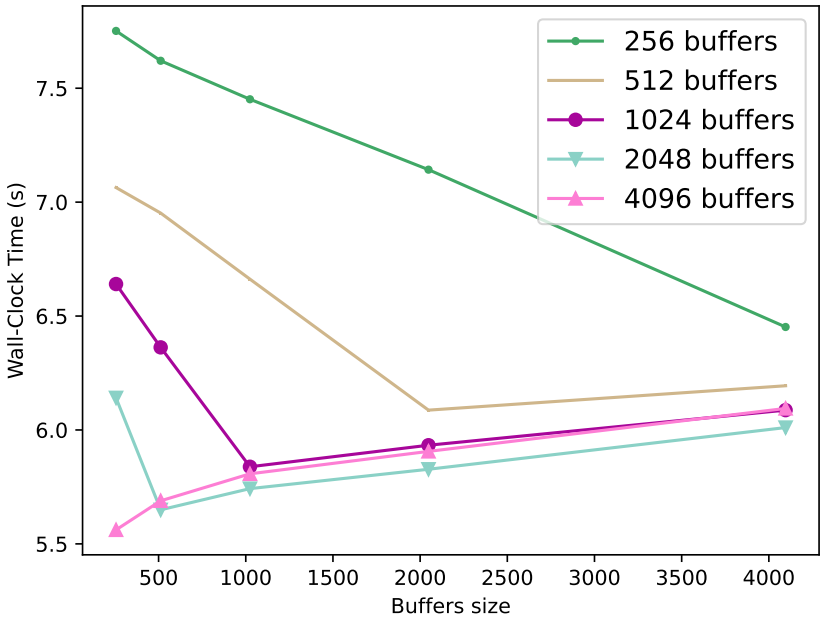


Figure 6.29: Simulation time of the PARSEC `water_nsquared` with variations of number of buffers and buffer size with **64** vCPUs

number of buffers and each buffer of a small size. The benchmark `water_nsquared` exhibits 4096x256 as the best configuration for all number of virtual CPUs.

6.4.2 Statistics validation

Mono-core: PolyBench/C suite

The statistics validation in mono-core was as simple to do as for our L1i model. Since the execution is deterministic, we made sure that we have exactly the same number of memory accesses and the same number of misses as the existing plugin.

Multi-core: PARSEC suite

Similarly, the statistics validation was simple to do in multi-core. In our L1d model, the only difference with the existing cache plugin is that we delay the simulation of the data cache in a separate thread. The cache plugin and our plugin retrieve all the memory accesses thanks to the only existing callback function of the TCG plugin API. So, when we ran all the 10 PARSEC benchmarks, we unsurprisingly found practically (due to non-determinism of multi-core execution) the same statistics with our plugin and the existing one.

6.4.3 Simulation time

Figure 6.30 shows the mean execution time for each program of the PolyBench/C suite in bare metal executed in QEMU. Our L1d model is implemented in a plugin named `dcachethread` with 1024 buffers of each a size of 1024. The simulation times of the y axis are on a log scale. We can see with disappointment that our L1d model is not faster than the existing cache plugin. Even if we notice a slight improvement for some benchmarks with our plugin (like `adi` and `correlation`), globally our L1d model executes with the same time as the existing plugin.

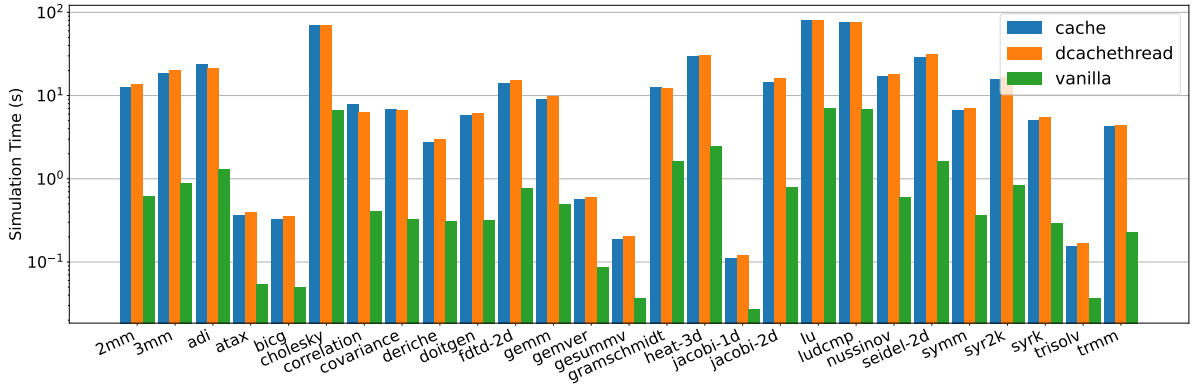


Figure 6.30: Simulation time of the PolyBench/C programs (log scale on y-axis), dcachethread with **1024x1024**

Regarding the results in multi-core of our L1d model, we just saw with the benchmark `water_nsquared` that the optimal combination of values of number and size of buffer is the same when increasing the number of virtual CPUs. We executed `water_nsquared` with the data cache of the existing cache plugin and with our L1d model with the optimal values number x size: 4096x256.

	cache	dcachethread 4096x256	dcachethread 8192x128
1 vCPUs	6.09	4.53	4.59
2 vCPUs	8.01	5.11	5.19
4 vCPUs	4.36	5.25	5.65
8 vCPUs	6.57	5.33	5.55
16 vCPUs	4.49	5.78	6.24
32 vCPUs	2.70	6.16	6.77
64 vCPUs	1.90	6.46	7.11

Table 6.7: Recap of `water_nsquared` mean execution time in seconds with L1d simulation in user-mode.

Table 6.7 summarize the mean of execution time on 20 executions of `water_nsquared`. We are faster than the existing cache plugin for 1, 2 and 8 virtual CPUs but for the rest, our L1d model degrades the performance a lot. We also execute our L1d model with the values 8192x128 and we can see that it does not help at improving the performance even if like with 4096x256, we are faster for 1, 2 and 8 virtual CPUs. As a result, 4096x256 seems to be the limit.

In conclusion, our L1d model can have a non-negligible improvement on the execution time for some benchmarks in multi-core. However not all the benchmarks in mono-core present such improvement and the values (number x size) of buffer and the number of virtual CPUs have an important impact on the simulation time. Regarding the multi-core execution, finding the optimal combination of values needs to be done by investigating each benchmark.

6.5 Cache hierarchy per virtual CPU: L1i + L1d + L2

In this section we do the evaluation of a cache hierarchy per virtual CPU composed of our L1i model with a simulation per TB, a naive L1d model (which is the same as the one in the existing QEMU `cache` plugin) and a L2 model on top of it. The L2 implementation per virtual CPU is described Section 5.3.2. Because of the issue raised by our L1i model, we used again the 10 benchmarks of the PARSEC suite with QEMU in `user`-mode. Here again, each experiment was run 20 times. In all the Figures in this section, we use `cache` to refer as the existing cache plugin in QEMU, `cacheTB` to refer to the plugin implementing our solution of the cache hierarchy and `vanilla` to refer to QEMU without any plugin activated. All the following experiments are done with these arbitrary cache configurations:

- L1i: 8-way, 32-set, 64 bytes per line and LRU
- L1d: 8-way, 32-set, 64 bytes per line, LRU and Write Back Allocate
- L2: 16-way, 64-set, 64 bytes per line, LRU, Non Inclusive Non Exclusive

6.5.1 Statistics validation

Mono-core: PolyBench/C suite

Like for the L1i evaluation, we run the suite in bare-metal in QEMU and since the behavior is deterministic, we make sure that the L2 statistics produced by our plugin are the same as the ones produced by the existing plugin of QEMU.

Multi-core: PARSEC suite

For all the 10 benchmarks, we computed the mean on the 20 executions of the total number of L2 accesses and the total of L2 misses. We varied the number of virtual CPUs with the following values: $\{1, 2, 4, 8, 16, 32, 64, 128\}$. We also added on each Figures a vertical black line for each value of virtual CPU that represents the range of 20 values.

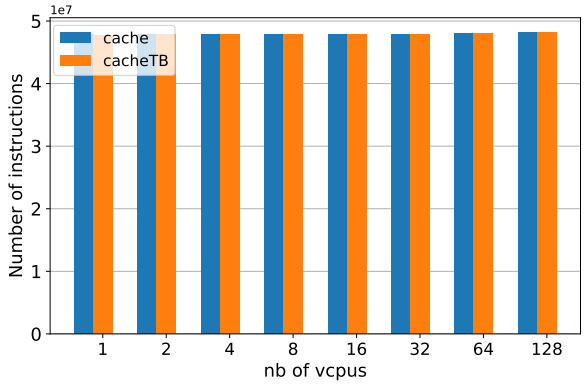


Figure 6.31: Total **number of L2 accesses** for lu_cb (log scale on x-axis)

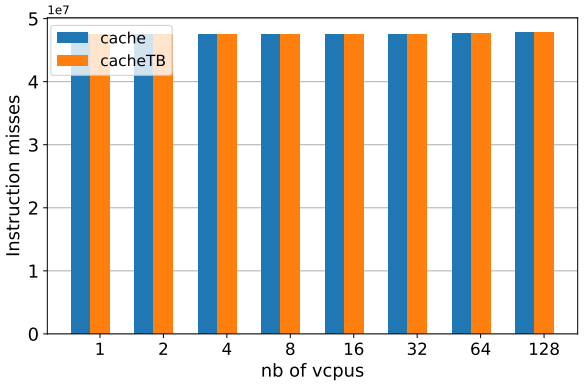


Figure 6.32: Total of **L2 misses** for lu_cb (log scale on x-axis)

Figure 6.31 and Figure 6.32 represent respectively the total number of L2 accesses and the total of L2 misses for the benchmark `lu_cb`. Appendix D contains the Figures of the statistics for the other benchmarks. The two Figures look similar because with the cache configurations that we chose and because of the characteristics of the benchmark,

we have a lot of misses in the L2 (more than 90%). However, it is not the case for all the other benchmarks as one can see in the Appendix. The vertical black lines are almost not visible on both Figures which shows that we have a very small variation of the values across the 20 executions. As the L2 statistics are mostly the same with our `cacheTB` plugin and the `cache` plugin, we can conclude on the good behavior and the validity of the produced statistics in multi-core.

6.5.2 Simulation time

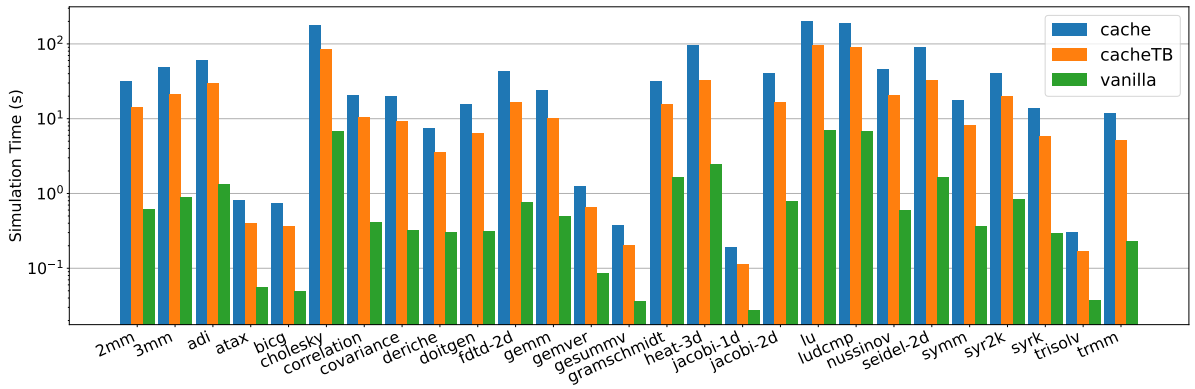


Figure 6.33: Simulation time of the PolyBench/C programs (log scale on y-axis)

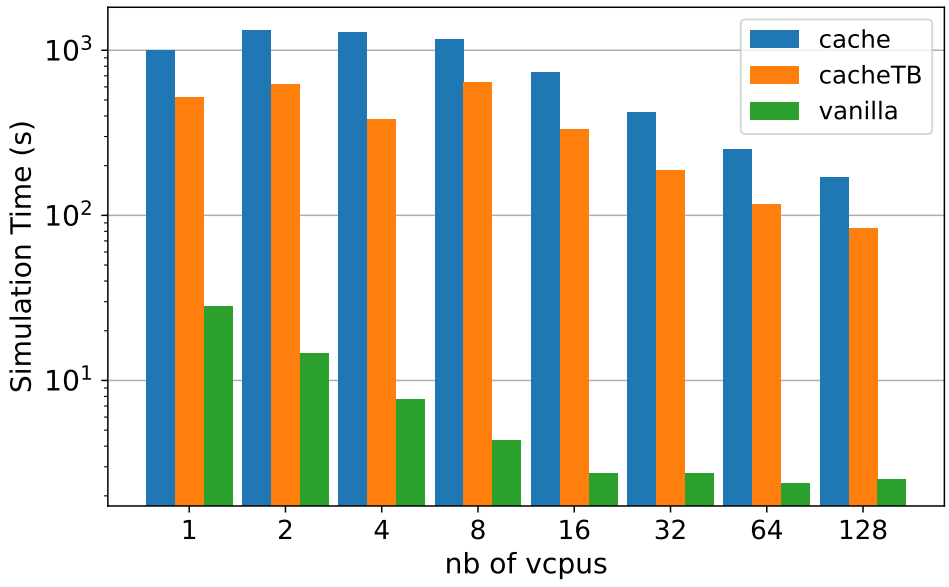


Figure 6.34: Simulation time of lu_cb (log-log scale)

We compare in this section the execution time of our `cacheTB` plugin composed of a L1i + L1d + L2 with the existing `cache` plugin. The difference of implementation with the existing plugin relies on the L1i model. The L1d and the L2 implementations are the same as the ones in the existing plugin. The idea in this section is to have an idea of the global improvement that we can have on the total cache hierarchy with only the L1i model that differs.

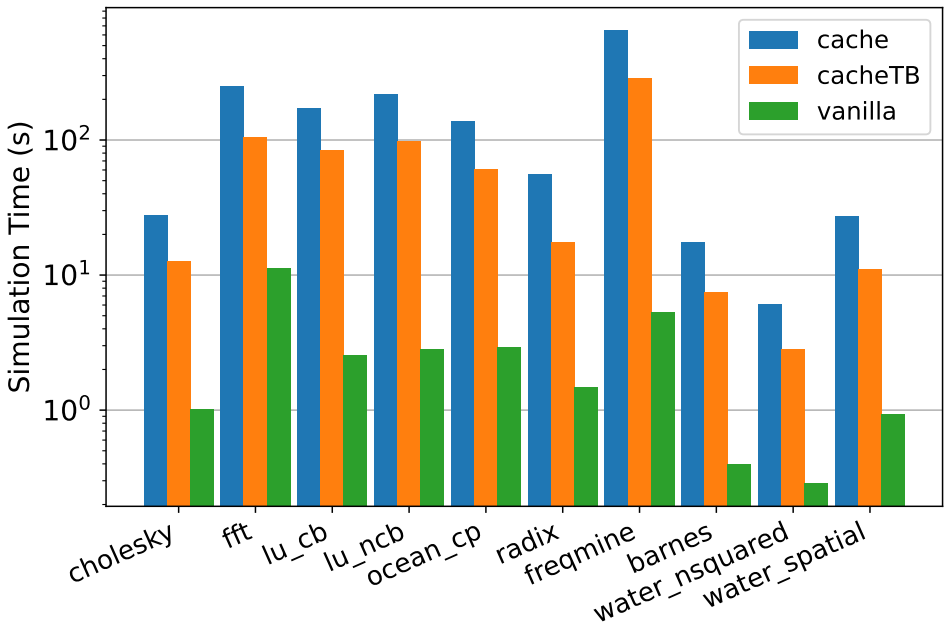


Figure 6.35: Simulation time of the PARSEC programs on 128 vCPUs (log scale on y-axis)

Figure 6.33 shows the mean of execution time for each program of the PolyBench/C suite in bare metal in QEMU with the y axis on a log scale. Figure 6.34 represents the simulation time of only the benchmark `lu_cb` and Figure 6.35 focuses on the simulation time with 128 virtual CPUs for all the 10 PARSEC benchmarks. Appendix D gives in detail the histograms of the simulation time for all the other PARSEC programs. One can see that our `cacheTB` plugin still remains faster but not as much as in Section 6.3.3.

	PolyBench/C	PARSEC
Speedup **cache** to **cacheTB**	2.19	2.43
Slowdown **cache** to **vanilla**	38.20	99.79
Slowdown **cacheTB** to **vanilla**	17.03	43.14

Table 6.8: Mean simulation time ratios.

Table 6.8 sums up all the different mean speedup/slowdown ratios. Even if we do not achieve here the same speedup as in Section 6.3.3, we are still faster than the existing cache plugin of a factor of 2 for the PolyBench and the PARSEC benchmark suites. To resume, with two levels of cache per virtual CPU, we have a reduction of 50% on the execution speed on average for all the benchmarks. Moreover, the slowdown ratios are higher than in Table 6.6 which is normal as we simulate a complete L2 hierarchy compared to only a L1i model.

6.6 Cache simulation with pinning

In this section, we wanted to see if using the pinning with our L1i model can change the behavior. The pinning works only in `full-system` and since we have the dependency on simulation runtime issue in `full-system`, using the pinning implementation might affect the behavior. To do this experiment, we used QEMU with busybear linux and our

plugin `cacheTB` and we ran inside the PARSEC benchmark `lu_cb`. We tested with the following virtual CPUs values: $\{2, 4, 8, 16, 32, 64, 128\}$. We used the following L1i configuration: 8-way, 32-set, 64 bytes per line and LRU. Table 6.9 sums up the execution time mean on 20 executions.

	cacheTB no pinning	**cacheTB pinning**
2 vCPUs	126.52	79.74
4 vCPUs	64.17	47.34
8 vCPUs	83.14	27.15
16 vCPUs	79.69	24.75
32 vCPUs	45.48	17.60
64 vCPUs	29.92	18.12
128 vCPUs	182.80	178.10

Table 6.9: Mean execution time in **seconds** of `lu_cb` in busybear without and with pinning and our `cacheTB` plugin.

Interestingly, using the pinning mechanism on top of our L1i model improves the performance. The execution time is considerably reduced. Regarding the cache statistics, the number of instructions and misses without and with pinning are close but not close enough to conclude on the correct behavior. There are still non-negligible differences.

In the end, we cannot draw a general conclusion. We just give here an overview of what happens when we combine our two contributions.

6.7 Conclusion

In this section, we studied and analyzed our contributions on how to improve the DBT engines simulation time and how to design smartly a cache model and a cache hierarchy per virtual CPU. All our contributions were done using the well-known DBT simulator QEMU.

First, we analyzed the scalability of QEMU parallel implementation which the methodology is described Chapter 4. In addition, we added to our methodology the possibility to pin the threads representing the virtual CPUs of QEMU on the host machine. For this contribution we have simulated a RISC-V and ARM target system up to 128 CPUs. Our host machine was composed of 32 physical CPUs and we relied on busybear Linux and the PARSEC benchmark to run the experiments. In conclusion of this study, QEMU parallel implementation provides a good scalability. However, much to our disappointment, bypassing the host Linux scheduler and forcing the virtual CPUs of QEMU to execute on a given set of physical CPUs does not have any effect.

Secondly, we studied the behavior and the performance of our L1i cache model described Chapter 5. The performance of a functional simulator is in general degraded by the addition of instrumentation. By taking advantage of the per TB execution of the DBT mechanism, we thought of a L1i simulation strategy that minimizes the overhead of using instrumentation. We raised two issues, one related to the DBT functioning that we can solve easily and another that is still unsolved on the time dependency. However, by only doing our analysis using QEMU in `user`-mode, we were able to conclude on the good behavior of our L1i model by validating the statistics against the existing cache

plugin. Our L1i model also shows considerably better performances (7 to 10 times faster than the existing cache plugin).

Thirdly, we analyzed the performances of our threaded L1d simulation presented Chapter 5 which are mitigated as it depends on the benchmark and the number of virtual CPUs. However, we outlined significant improvement with our model on specific number of virtual CPUs for the values 4096x256.

Fourthly, we studied a simple cache hierarchy per virtual CPU composed of a L1i, L1d and L2 on top of it. Our contribution in the cache hierarchy relies only on the L1i model. The L1d and L2 implementations are the same as the ones in the existing plugin. We can achieve a factor 2 of improvement compared to the existing cache plugin.

As the last section of this chapter, we decided to combine two of our contributions. We used the pinning mechanism on top of our L1i simulation and interestingly, the performances are improved when using the pinning but the cache statistics are unstable.

Chapter 7

Conclusion and Prospects

THIS thesis deals with the topic of how to best exploit Dynamic Binary Translation for the simulation of software centric systems. Although this simulation mechanism is known for its good simulation speed, investigating on how to improve it even more raises questions that were introduced in Chapter 2 and to which we come back to in this conclusion.

How can we assert that DBT parallel implementation scales well on the multi-core host machine?

We presented in Chapter 4 an experimental methodology to study the parallel implementation of the well-known DBT-based simulator QEMU. Even though the support for parallel execution has been around for some time, no comprehensive performance study had been performed before to the best of our knowledge. The idea was to assert its good scalability. To do so, we have tested a multitude of parameters to aim for the most objective evaluation. Our host machine is composed of 32 physical CPUs and we experimented with up to 128 virtual CPUs. Therefore, with all the results, we concluded that QEMU parallel implementation as it is scales well. Moreover, there is one case where we have a greater scalability: when disabling the SMT parameter meaning only one CPU runs per core instead of two. But disabling the SMT parameter has a drawback: it divides by 2 the total number of physical CPUs on the host and thus limits parallelism.

Can we rely on the host configuration to improve the DBT parallel simulation speed?

Still in Chapter 4, we investigated whether using thread affinity of QEMU's virtual CPUs on the host CPUs can reduce the global execution time when using a multi-core host. Using this mechanism, named pinning, consists of forcing a virtual CPU of the target to run exclusively on a given host CPU. By bypassing the Linux scheduler and forcing each thread of the virtual CPUs of QEMU to run on chosen host CPUs, we thought we could achieve positive results. Moreover, to achieve the best possible results, we did not decide the virtual CPUs thread assignation randomly. Choosing the host CPU to force the execution of each thread was done in a smart way by following the NUMA nodes distribution of the host. Incrementally, we filled the nodes one by one. To be able to objectively conclude on the results, two target ISAs were used in this study: RISC-V mainly motivated by the wide adoption of this ISA in research and in industry and the ARM ISA. When performing all the experiments on both targets, to our amazement, pinning the virtual CPUs of QEMU does not affect the simulation time.

In the end the Linux scheduler cannot be surpassed by replacing it with the pinning mechanism.

The other two questions were more challenging to answer. They deal with the topic of Dynamic Binary Instrumentation. Adding instrumentation in functional simulation, such as in QEMU, will induce a non-negligible execution time overhead. Because we did not want to modify intrusively QEMU ourselves and because a framework to do instrumentation in QEMU is available since version 4.2, we decided to use this framework to build our cache model contributions on. This framework is named Tiny Code Generator (TCG) Plugins and provides a basic API to easily and efficiently instrument QEMU.

Can we benefit from the DBT approach to design in particular a cache model that limits the impact on the global simulation time?

We proposed in Chapter 5 an instruction cache model that relies on the per block nature of the DBT. Our intuition was that thanks to this DBT principle, we can know in advance, meaning at the translation stage of the blocks just before execution, which instructions in a block will be a hit for sure and which might miss. With this information, we only need to run the instruction cache simulation on those instructions that might be a miss and thus drastically reduce the overhead compared to a naive cache simulation that would run for each and every instruction. Our instruction cache model works with a multi-core simulation as the blocks in the DBT are executed by the threads representing the virtual CPUs. For all the experiments performed, we compared our results with the existing cache plugin that does a naive instruction per instruction cache simulation. We obtained the same statistics but at far better performances.

How can we enhance the time accuracy of the DBT mechanism when adding models of new architectural features in the simulation without overly degrading simulation speed?

We investigated in Chapter 5 a threaded model of a data cache. The new architectural feature in this contribution is the addition to QEMU of a functional data cache simulator. As adding new features in a simulator will for sure degrade the performance, we fulfilled a reasonable accuracy/speed trade-off by simulating a data cache model offline in a separate thread. The principle is to give each virtual CPU a list of buffers that they filled up with their memory accesses and once a buffer is full, it is put in a global list used by the separate thread to run the data cache simulation. Even if this data cache model shows some improvements compared to the existing cache plugin, it is application dependent and is only effective with a small number of virtual CPUs. When the number of virtual CPUs is high, a bottleneck happens on the data cache thread as all the virtual CPUs will want to access the global list of full buffers at the same time.

7.1 Prospects

Although the contributions of this thesis allow a better understanding of several DBT related topics, some points remain to be considered. This section explains the ideas to deal with those points.

Cache coherency

Our cache model presented Section 5.3.2 stops at a hierarchy per virtual CPU with a L1 instruction, a L1 data and a L2 on top on both. However, this model is not representative of what exists in modern computers. At least another level of cache is usually present in nowadays multi-core architectures to help improve the performance of the local L1 and L2 levels. Figure 2.1 is a good example were multiple CPUs share the same level of cache (L3 level or also called LLC for Last Level Cache) which is typical of multi-core architectures. Figure 7.1 is another arbitrary example where each two CPUs share the same L3 cache level.

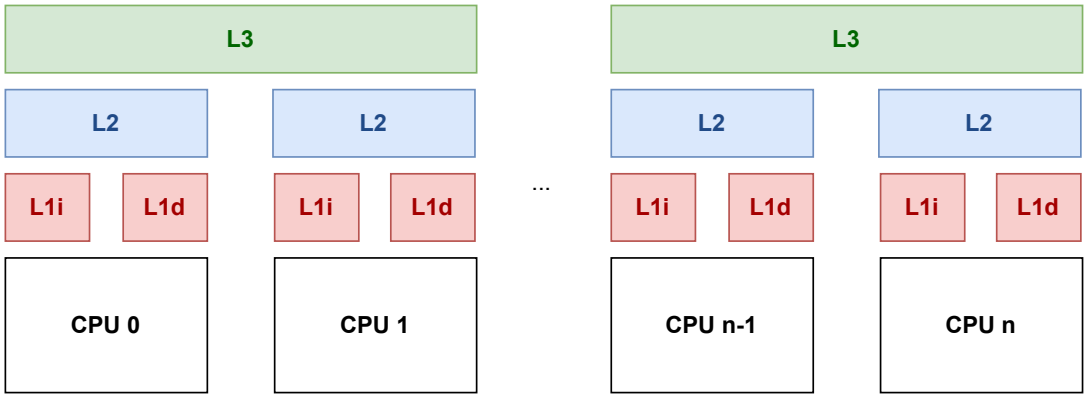


Figure 7.1: Arbitrary example of the cache hierarchy of $n + 1$ CPUs

To implement such a hierarchy, our L1d threaded implementation is no longer viable. Indeed, in the thread that does the data cache simulation, we parse buffer by buffer and each buffer belongs to only a virtual CPU. We do not recover the exact order of the memory accesses of all the virtual CPUs, we only have the chronological order inside a virtual CPU. Thus, it becomes a problem when adding a shared level since misses can happen if the accesses are reversed. To overcome this problem, one can think of adding a timestamp of a global date in order to rebuild the exact order of all the memory accesses.

Moreover, with systems like these, it might happen that many copies of a data exist among the cache levels (for example one copy is in the shared level and other copies are in the local level of the CPUs). If one of the copies of the data is changed locally, all the other caches are then in an incoherent state. The modification of the data must be passed on to the other caches that have a copy. To deal with this issue, different cache coherency protocols exist (MSI, MESI, MOESI,...) to ensure the good behavior of the global system. To propagate the change for all the copies, it can be useful to know which cache has which data. The naive solution for a simulator would be to browse all the caches. Another solution would be to have somewhere the information on which cache has which data and for that multiple storage structures can be used. The naive solution will definitely induce an important execution time overhead, that is why it needs to be done in a smart way.

We saw in this thesis that instrumenting DBT based engines such as QEMU has an important cost. Adding cache coherency in our cache model raises a performance problem as it will produce a huge overhead. To limit the impact, we cannot browse the entire local caches each time to check if they have the data. To remain at a functional cache simulation, multiple points can be considered. The first immediate idea is to store the pointer of the cache that has the looking data. Thus, we avoid checking all the caches. The other idea is to look in detail at the existing memory hierarchies to see

how they deal with cache coherency and how we can adapt their hardware approaches to our functional model. To this end, it is necessary to re-study the parallelization strategies like we have done for the contribution on the threaded data cache.

Dependency on QEMU runtime

The main remaining problematic issue related to our cache model contribution in QEMU is its dependency on runtime. Even if we managed to prove the correct behaviour of our cache model in QEMU `user`-mode, the `full-system` mode reflects more closely the reality and thus evaluating our cache model in this mode makes more sense. That is why succeeding in evaluating our cache model in `full-system` is a crucial point that must be addressed. We tried a simple patch to stop counting ticks during cache model execution with the hope to have much more stable statistics but sadly it was not a real success. For this point, a more intrusive research and a deeper understanding of QEMU time handling mechanisms is mandatory.

QEMU cache simulation and security

The last point left to be addressed is whether our cache model can be used for security research. Lots of attacks are known to be related to caches. It is the case of the two famous Spectre and Meltdown attacks. In the contributions of this thesis, we proposed a fast and accurate simulation of a simple cache model. Some attacks can deduce by the time spent in the cache if there is a miss or a hit (a miss takes more time) and therefore retrieve information. By improving our cache model with a pseudo timed cache simulation (adding some arbitrary time for each miss), we can propose a fast and accurate functional cache simulation interesting for the analysis of attacks.

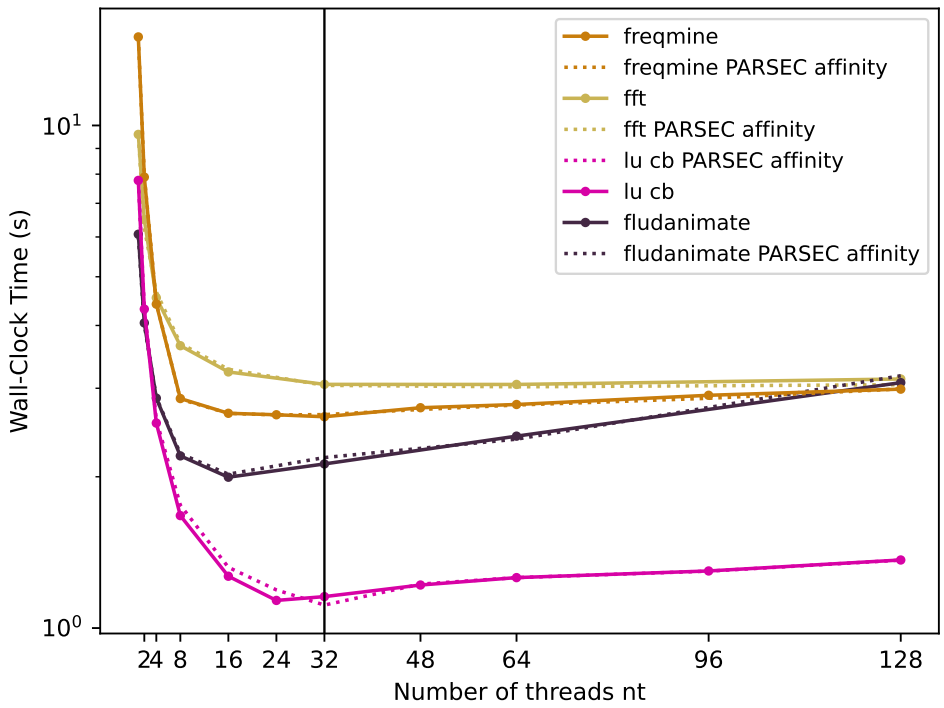
APPENDIXES

Appendix A

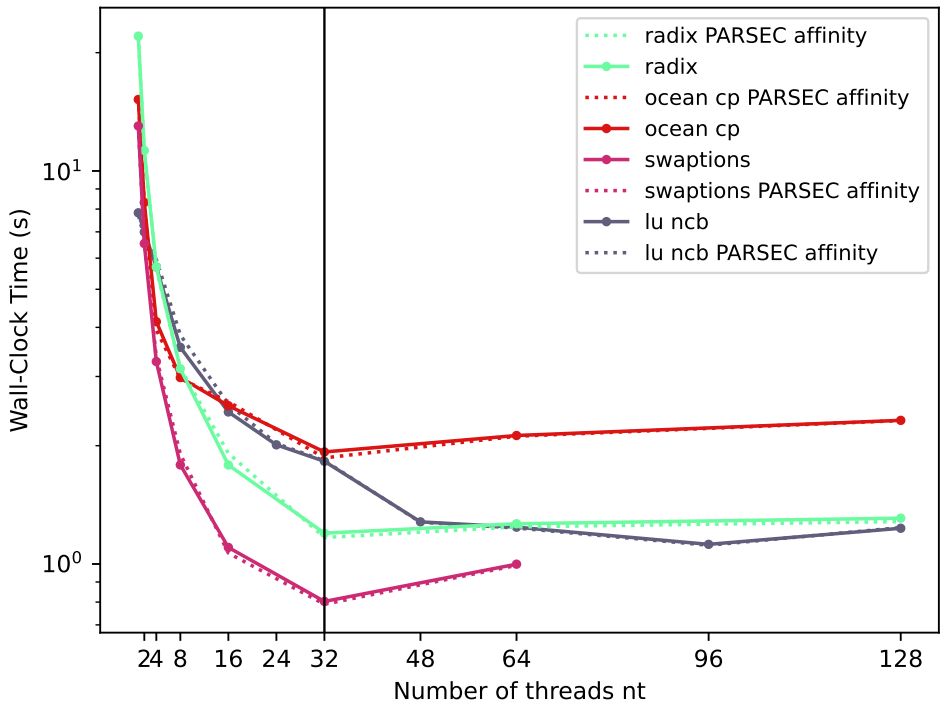
Scalability of QEMU Parallel Implementation

This Appendix contains the detailed comparisons for all the benchmarks that are not in Section 6.2 regarding the scalability of QEMU parallel implementation. The PARSEC benchmark `barnes` was very instable in the RISC-V simulation, that is why it appears on some Figures in x86 and ARM but is not present in the RISC-V Figures.

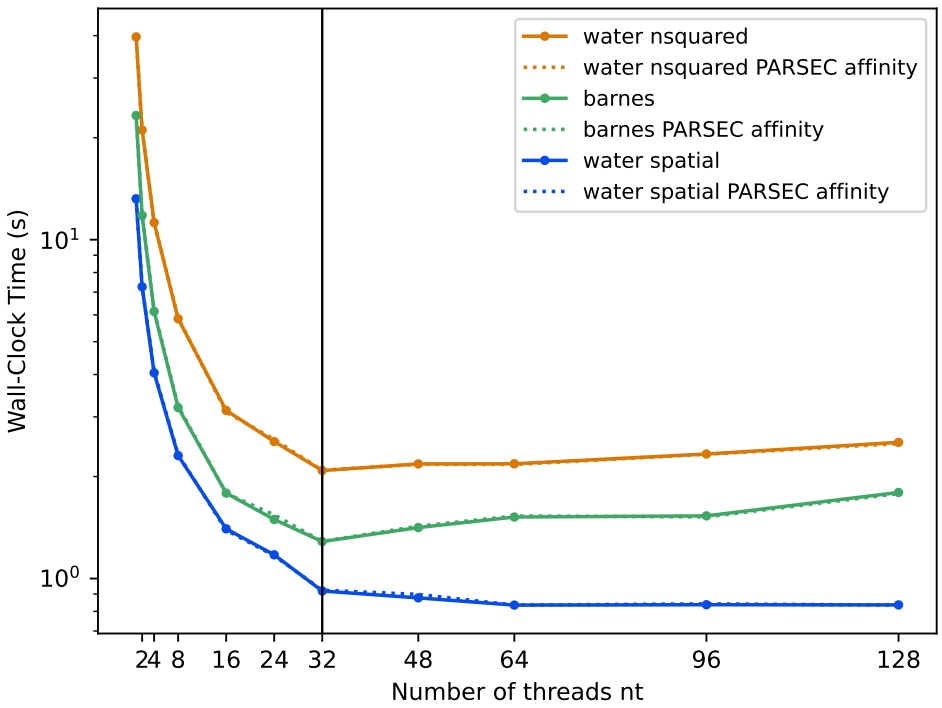
Figures



(a) Comparison full execution time in **x86** part 2 of the benchmarks

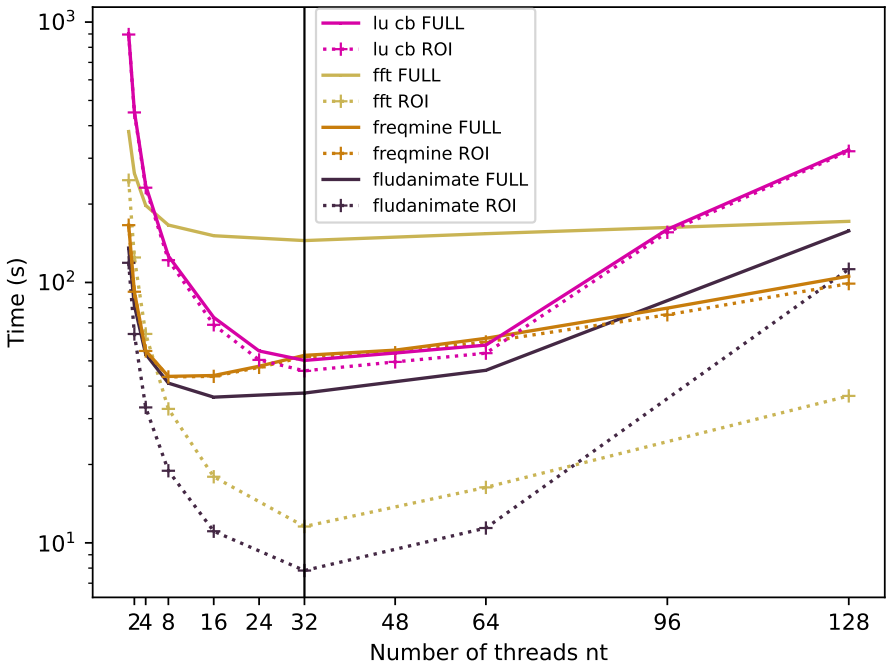


(b) Comparison full execution time in **x86** part 3 of the benchmarks

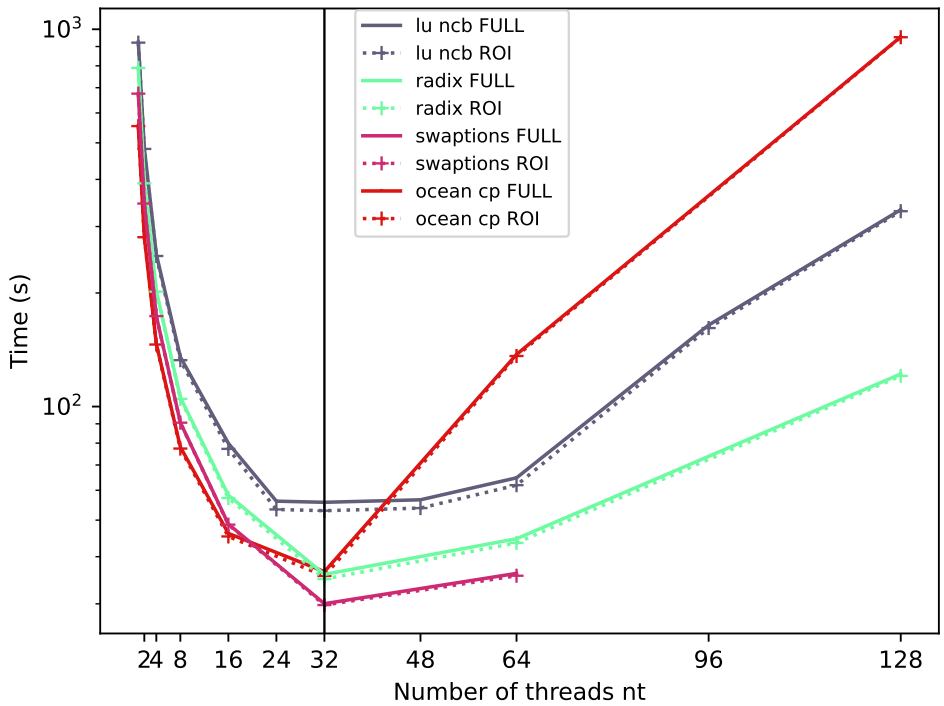


(c) Comparison full execution time in **x86** part 4 of the benchmarks

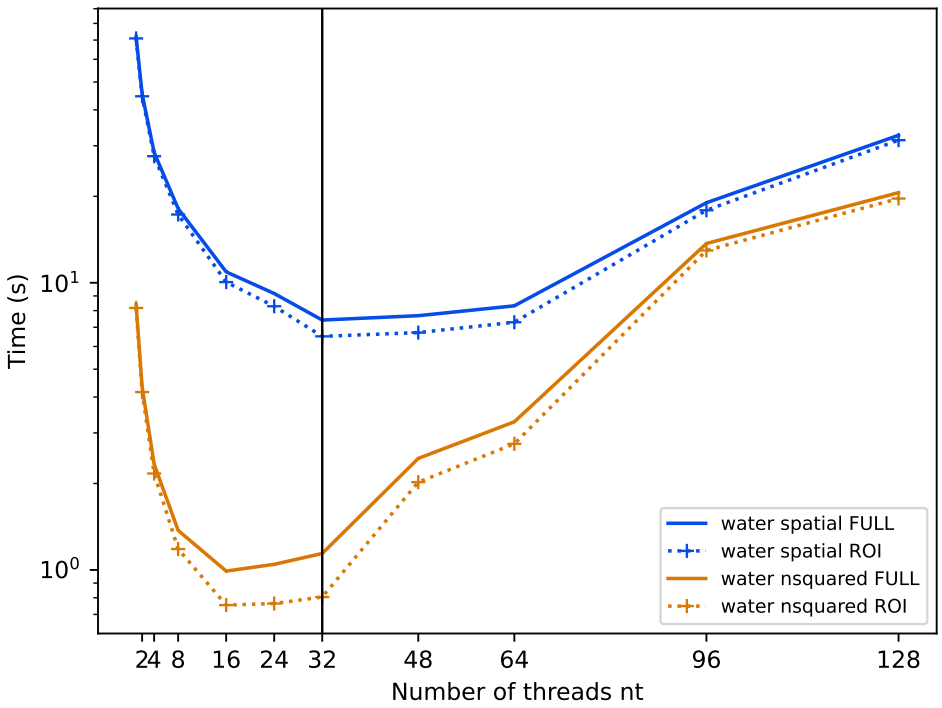
Figure A.1: Comparison of the full execution time with and without the PARSEC thread affinity in x86



(a) Comparison **full** and **ROI** execution time in QEMU **RISC-V** $n_c = n_t$ **without pinning** part 2 of the benchmarks

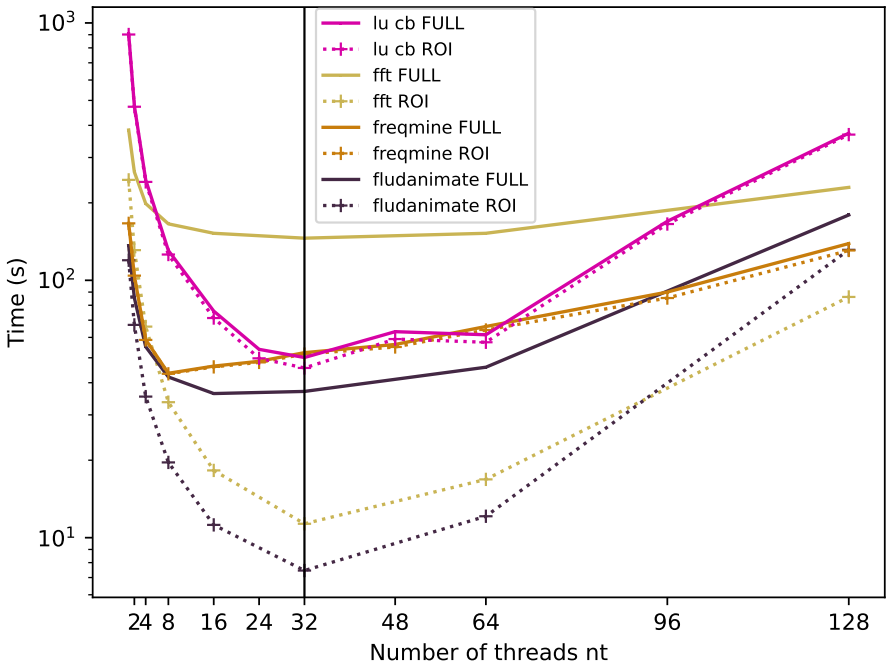


(b) Comparison **full** and **ROI** execution time in QEMU **RISC-V** $n_c = n_t$ **without pinning** part 3 of the benchmarks

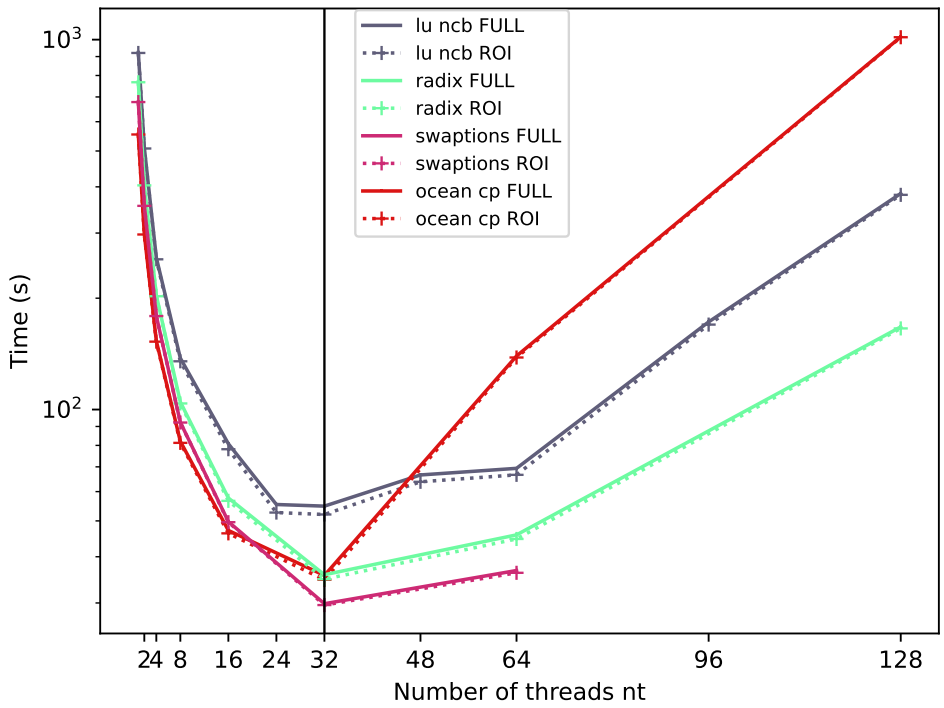


(c) Comparison **full** and **ROI** execution time in QEMU **RISC-V** $n_c = n_t$ **without pinning** part 4 of the benchmarks

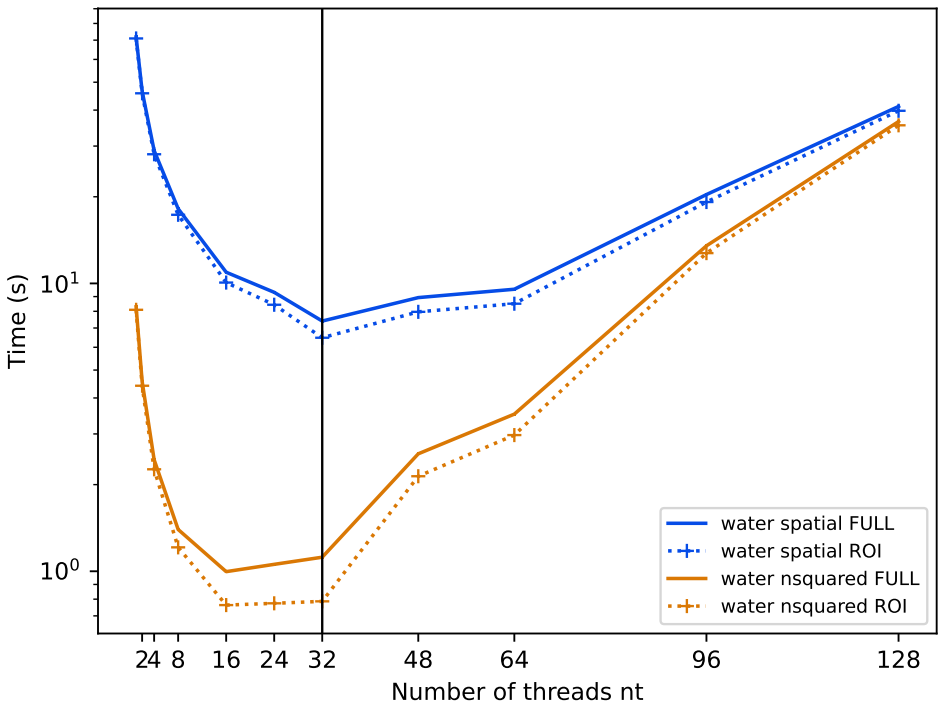
Figure A.2: Comparison of the **full** and **ROI** execution time in QEMU in **RISC-V** with $n_c = n_t$ **without pinning**



(a) Comparison **full** and **ROI** execution time in QEMU **RISC-V** $n_c = n_t$ **with pinning** part 2 of the benchmarks

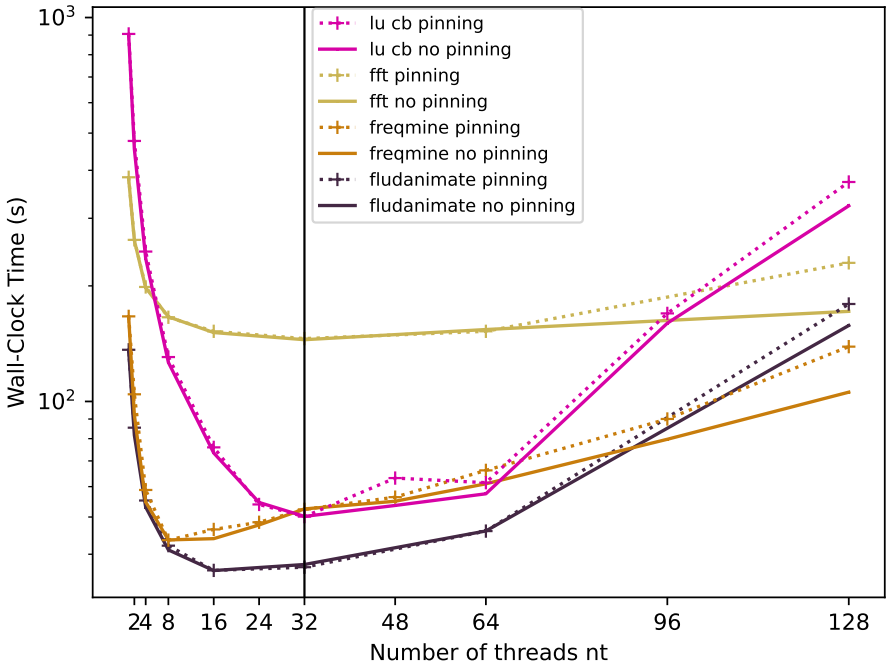


(b) Comparison **full** and **ROI** execution time in QEMU **RISC-V** $n_c = n_t$ **with pinning** part 3 of the benchmarks

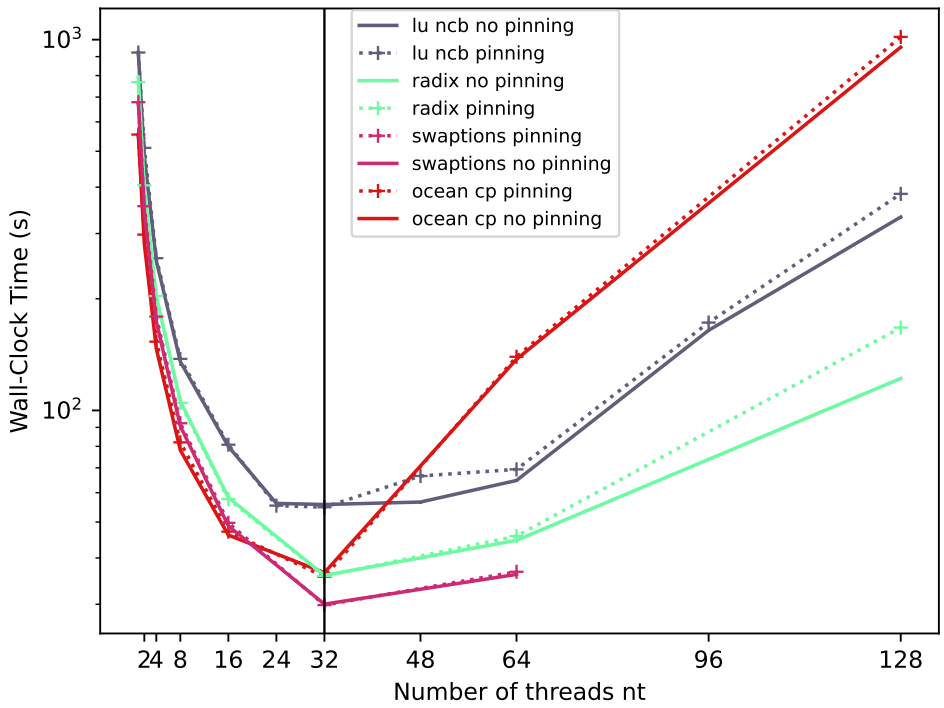


(c) Comparison **full** and **ROI** execution time in QEMU **RISC-V** $n_c = n_t$ **with pinning** part 4 of the benchmarks

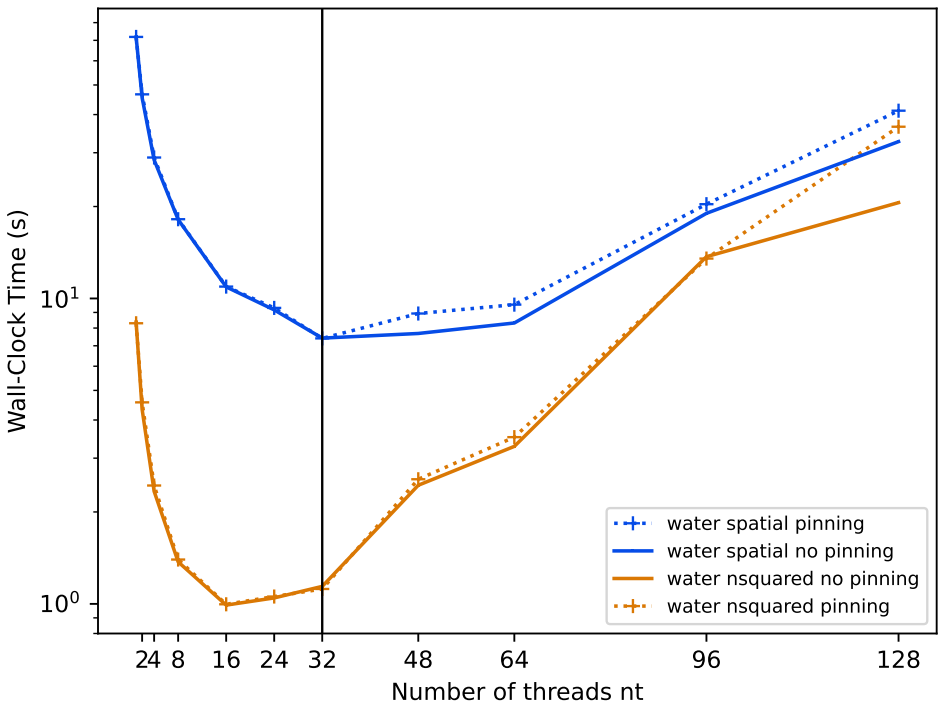
Figure A.3: Comparison of the **full** and **ROI** execution time in QEMU in **RISC-V** with $n_c = n_t$ **with pinning**



(a) Comparison **full** execution time in QEMU **RISC-V** $n_c = n_t$ **without pinning** and **with pinning** part 2 of the benchmarks

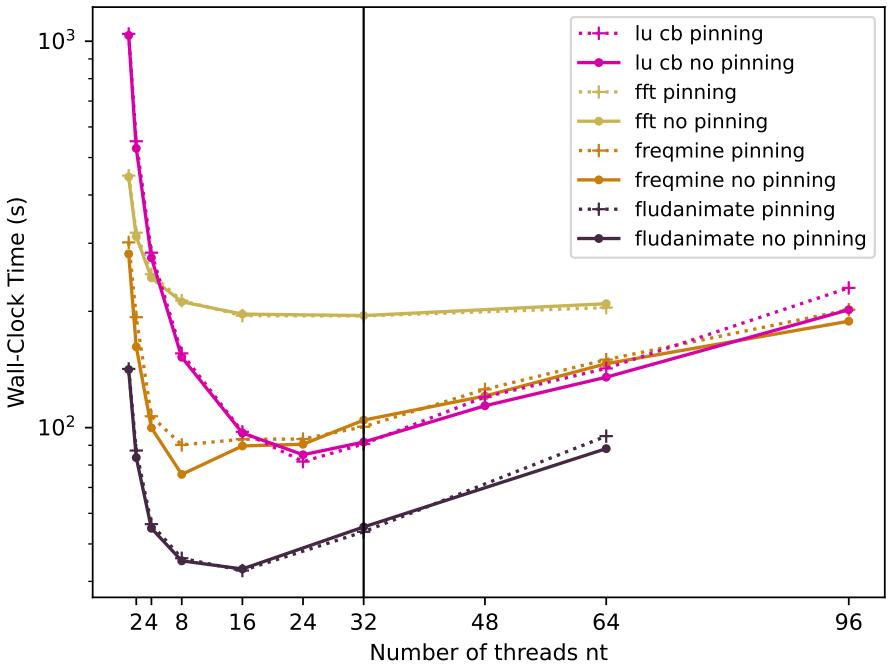


(b) Comparison **full** execution time in QEMU **RISC-V** $n_c = n_t$ **without pinning** and **with pinning** part 3 of the benchmarks

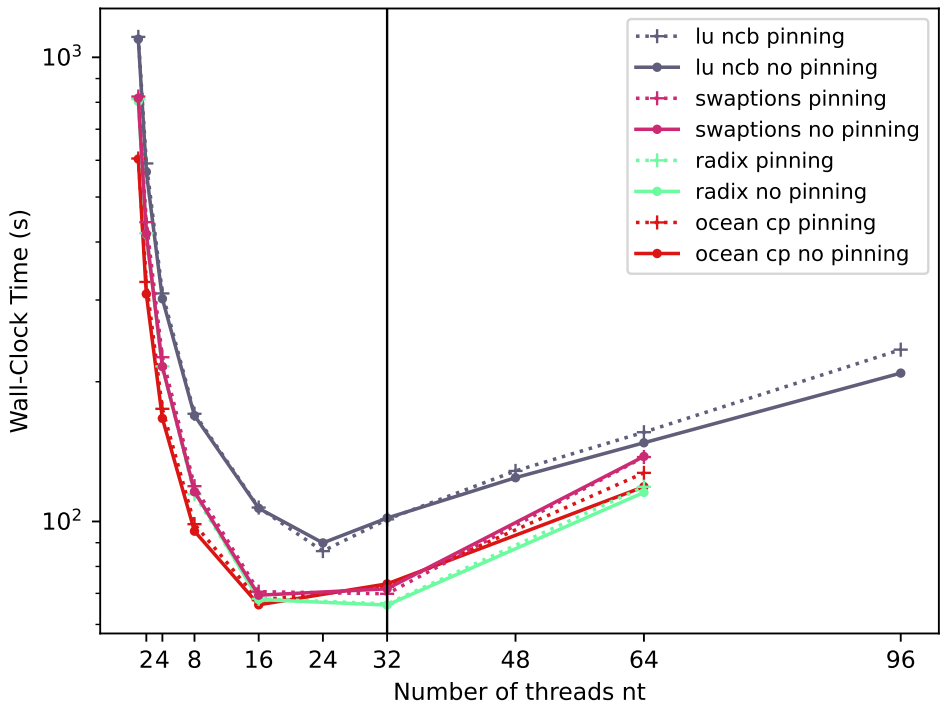


(c) Comparison **full** execution time in QEMU **RISC-V** $n_c = n_t$ **without pinning** and **with pinning** part 4 of the benchmarks

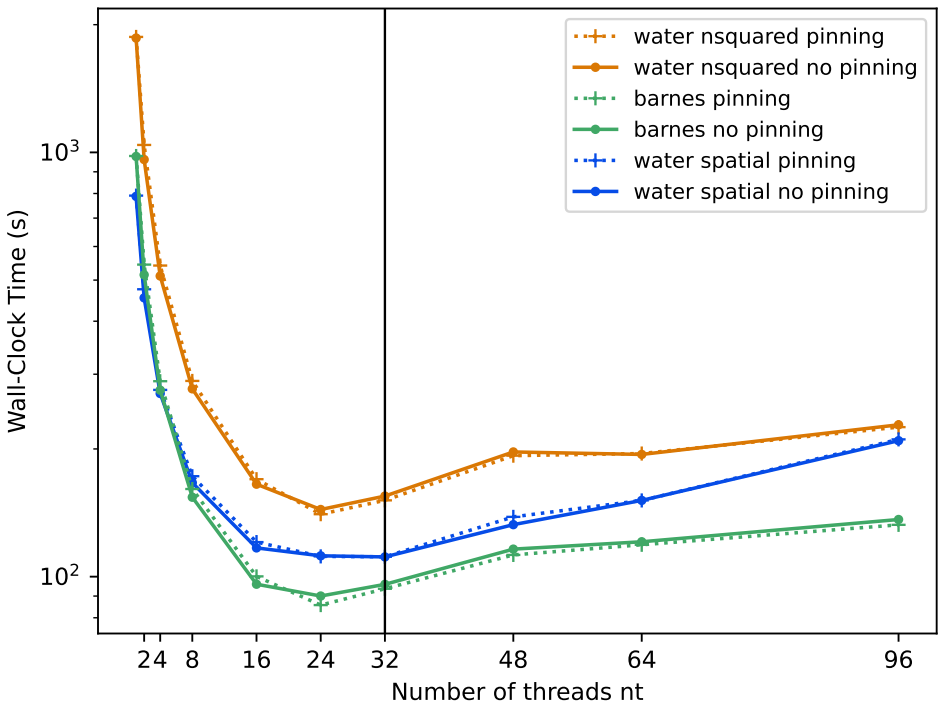
Figure A.4: Comparison of the **full** execution time in QEMU in **RISC-V** with $n_c = n_t$ **without pinning** and **with pinning**



(a) Comparison **full** execution time in QEMU **ARM** $n_c = n_t$ **without pinning** and **with pinning** part 2 of the benchmarks



(b) Comparison **full** execution time in QEMU **ARM** $n_c = n_t$ **without pinning** and **with pinning** part 3 of the benchmarks



(c) Comparison **full** execution time in QEMU **ARM** $n_c = n_t$ **without pinning** and **with pinning** part 4 of the benchmarks

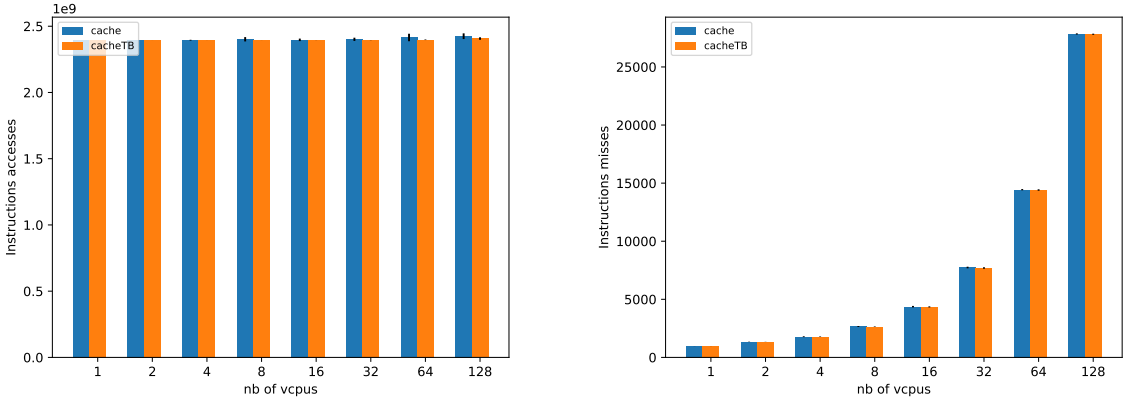
Figure A.5: Comparison of the **full** execution time in QEMU in **ARM** with $n_c = n_t$ **without pinning** and **with pinning**

Appendix B

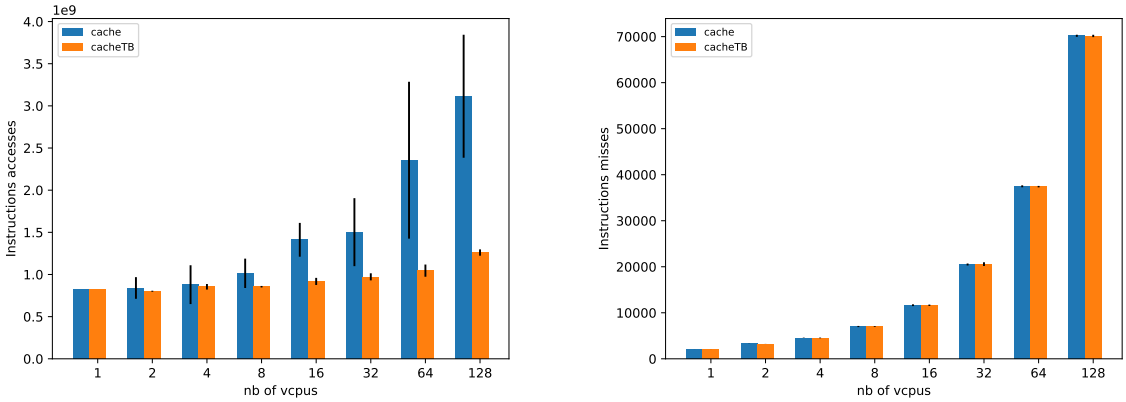
Instruction cache (L1i) evaluation

This Appendix contains the detailed results on the statistics and execution time for all the benchmarks that are not in Section 6.3.

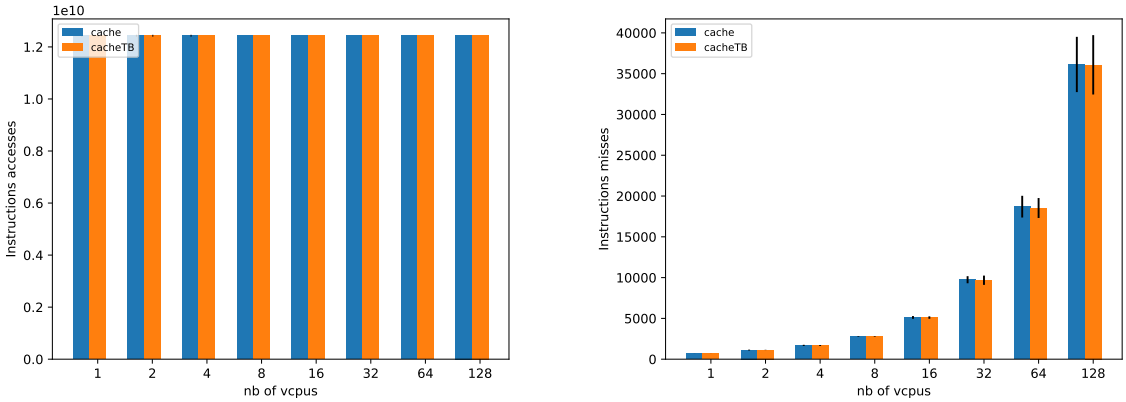
Figures



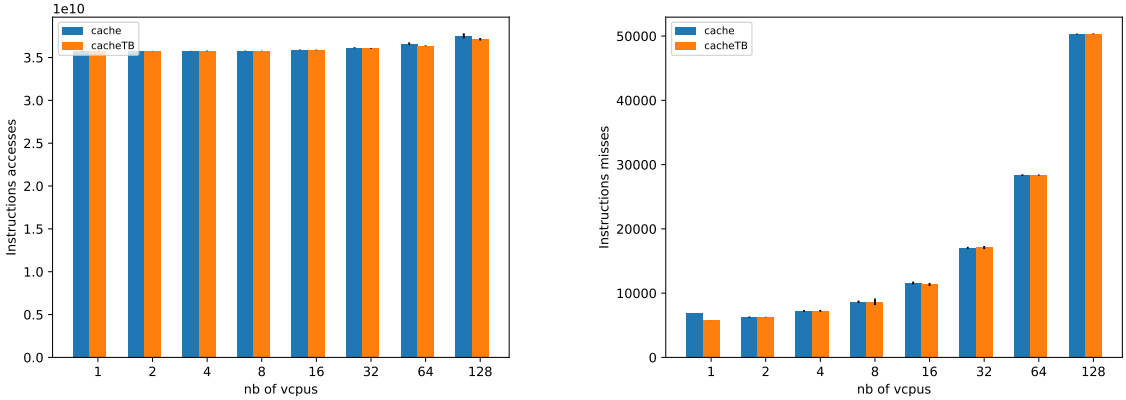
(a) Total **number of instructions** (left) and **misses** (right) for BARNES (log scale on x-axis)



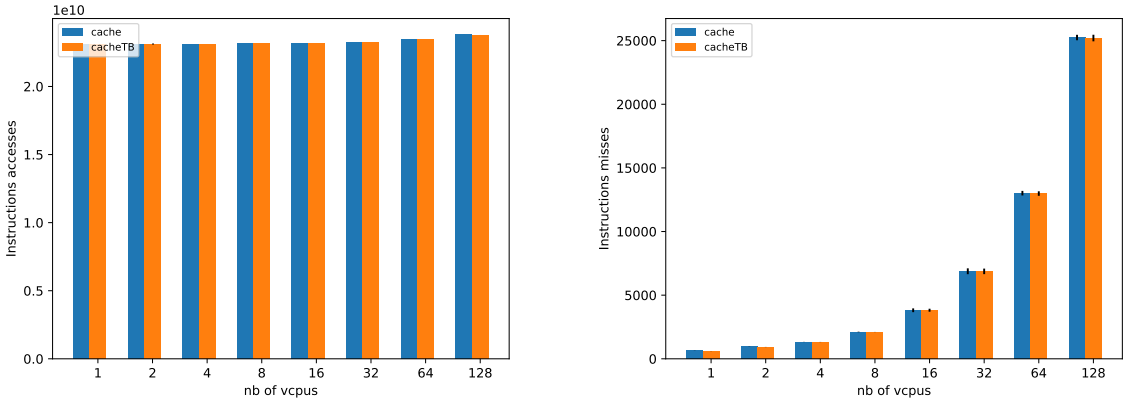
(b) Total **number of instructions** (left) and **misses** (right) for CHOLESKY (log scale on x-axis)



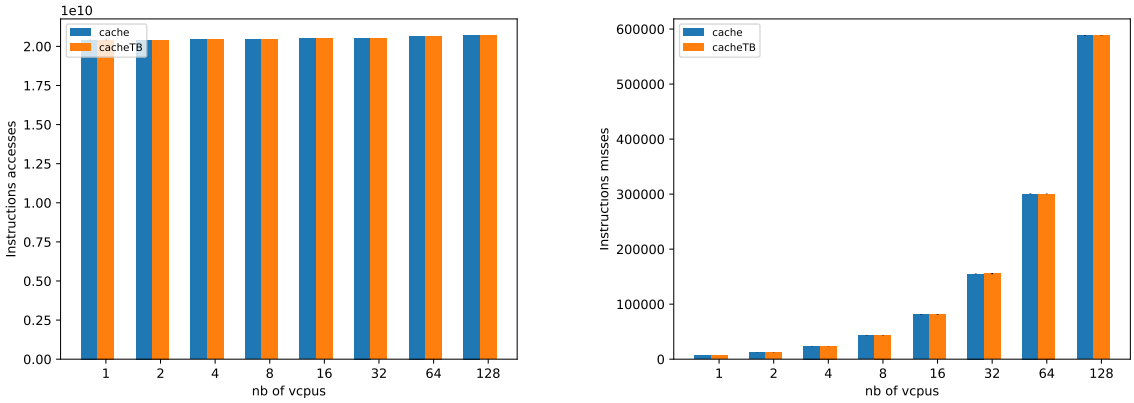
(c) Total **number of instructions** (left) and **misses** (right) for FFT (log scale on x-axis)



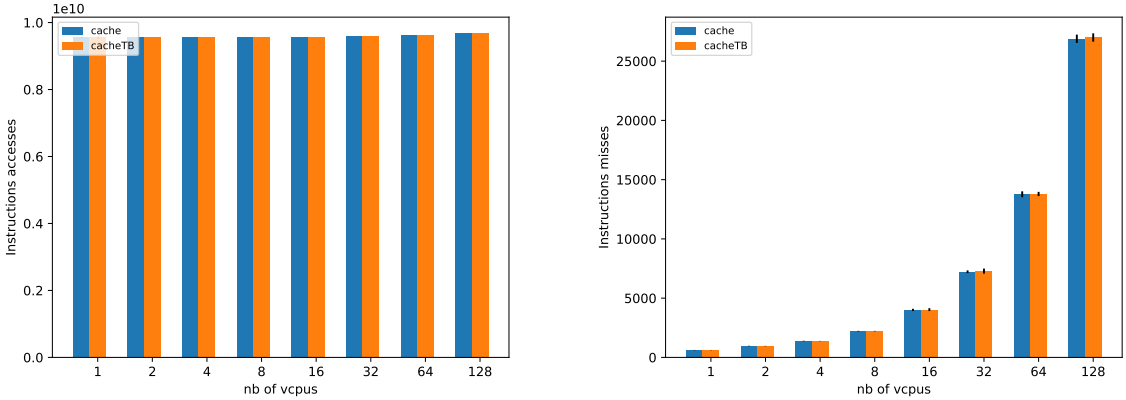
(d) Total **number of instructions** (left) and **misses** (right) for FREQMINE (log scale on x-axis)



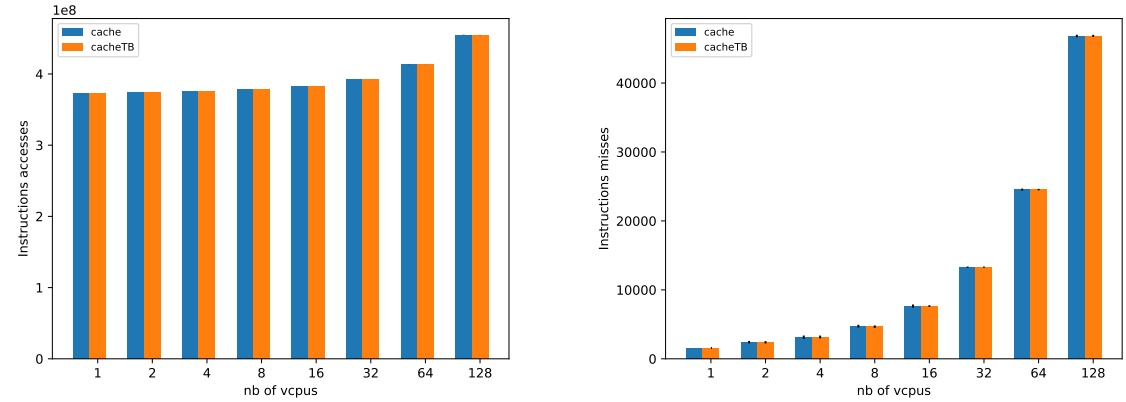
(e) Total **number of instructions** (left) and **misses** (right) for LU_NCB (log scale on x-axis)



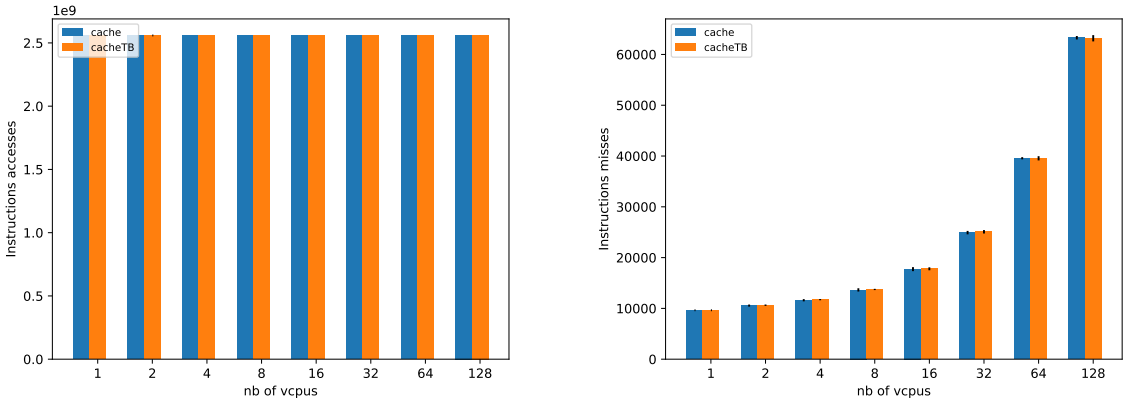
(f) Total **number of instructions** (left) and **misses** (right) for OCEAN_CP (log scale on x-axis)



(g) Total **number of instructions** (left) and **misses** (right) for RADIX (log scale on x-axis)

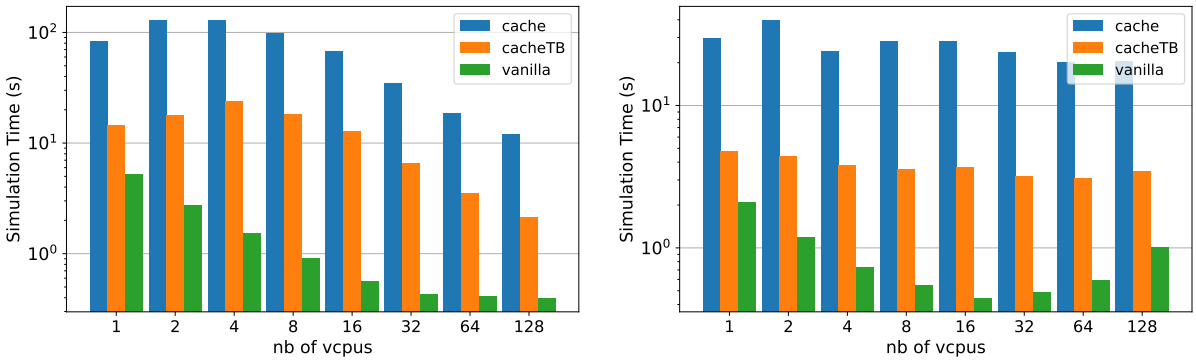


(h) Total **number of instructions** (left) and **misses** (right) for WATER_NSQUARED (log scale on x-axis)

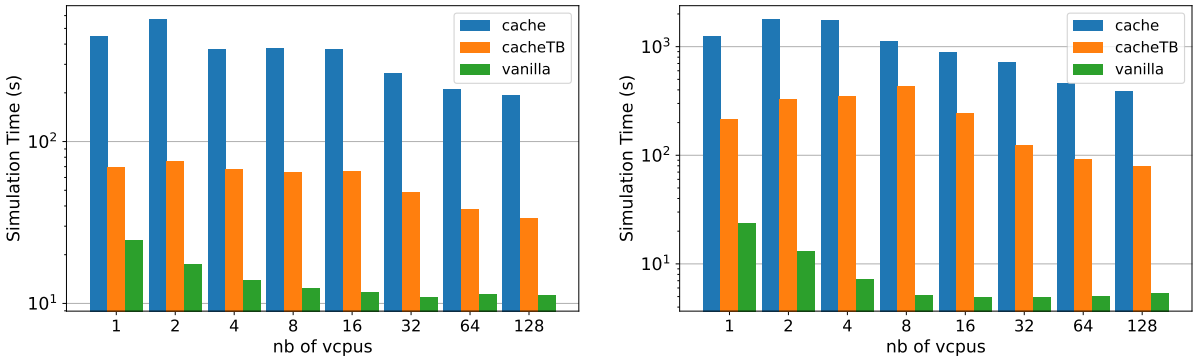


(i) Total **number of instructions** (left) and **misses** (right) for WATER_SPATIAL (log scale on x-axis)

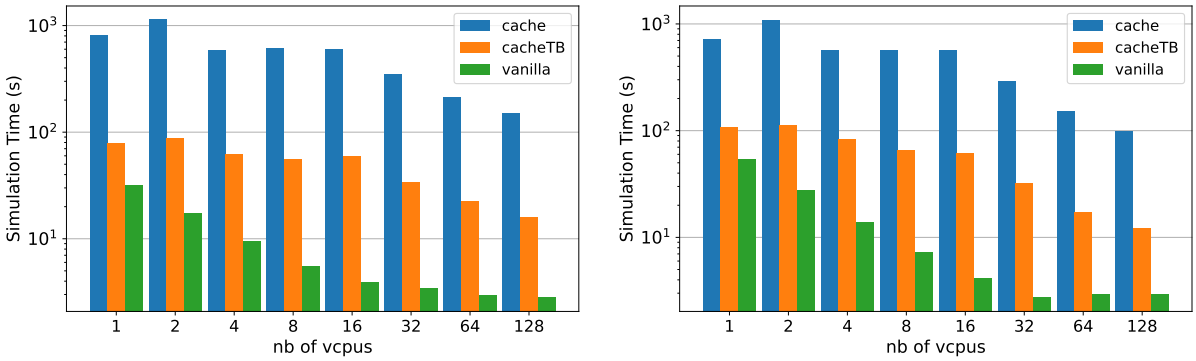
Figure B.1: Instruction cache statistics in QEMU `user`-mode of all the other PARSEC



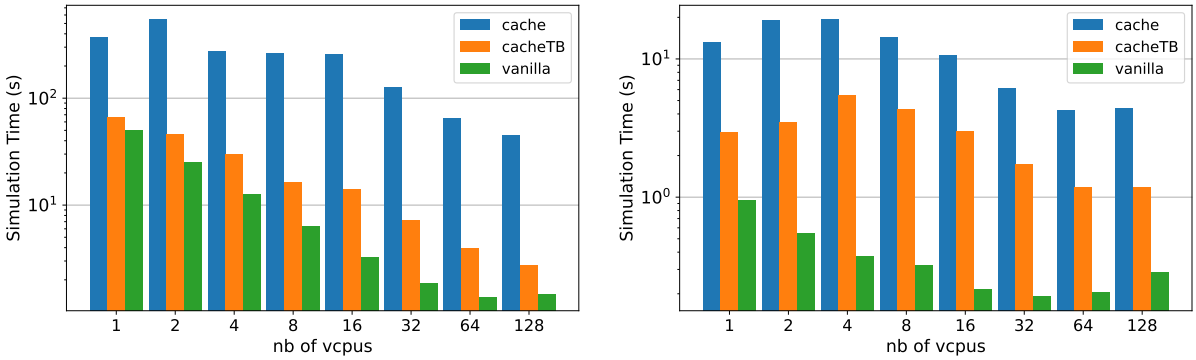
(a) Simulation time of BARNES (left) and CHOLESKY (right) (log-log scale)



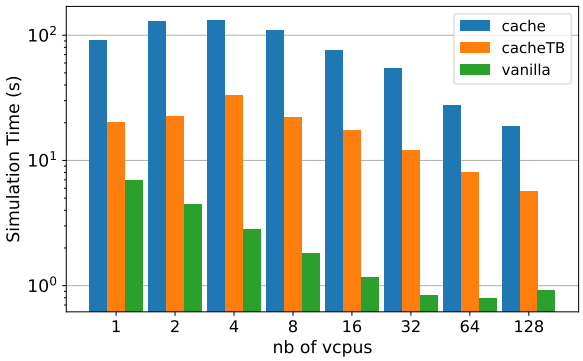
(b) Simulation time of FFT (left) and FREQMINE (right) (log-log scale)



(c) Simulation time of LU_NCB (left) and OCEAN_CP (right) (log-log scale)



(d) Simulation time of RADIX (left) and WATER_NSQUARED (right) (log-log scale)



(e) Simulation time of WATER_SPATIAL (log-log scale)

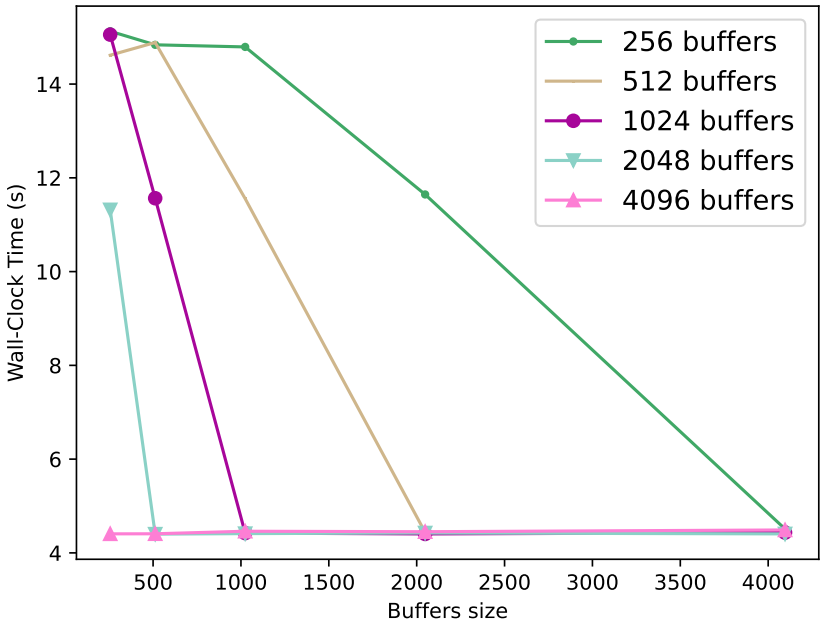
Figure B.2: Simulation time with instruction cache in QEMU `user`-mode of all the other PARSEC

Appendix C

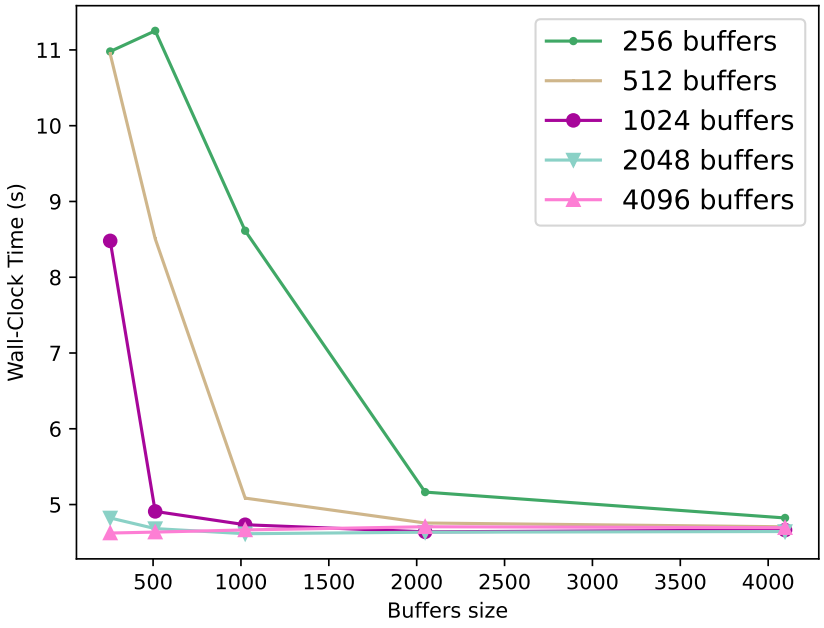
Data cache (L1d) evaluation

This Appendix contains the detailed results on the investigation of the optimal configuration of buffers for the `water_nsquared` benchmark that are not in Section 6.4.

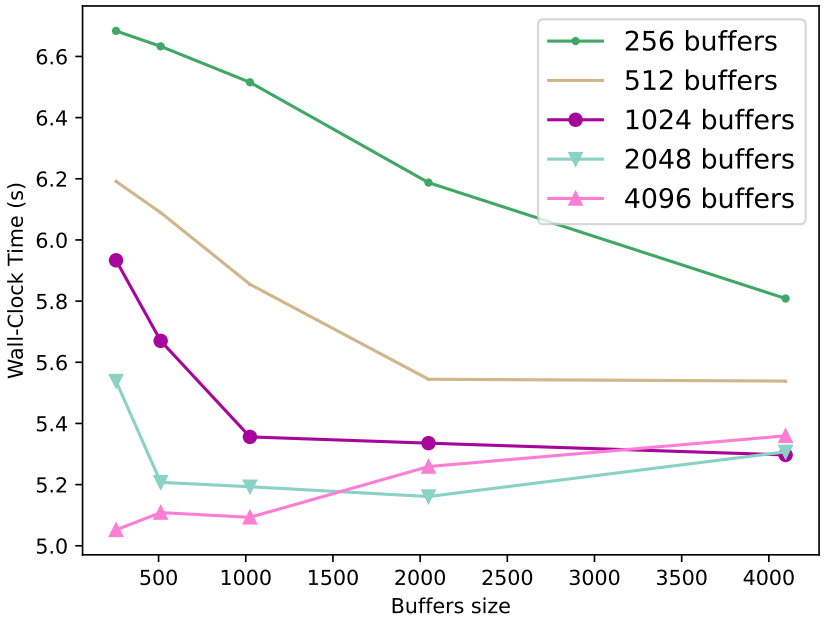
Figures



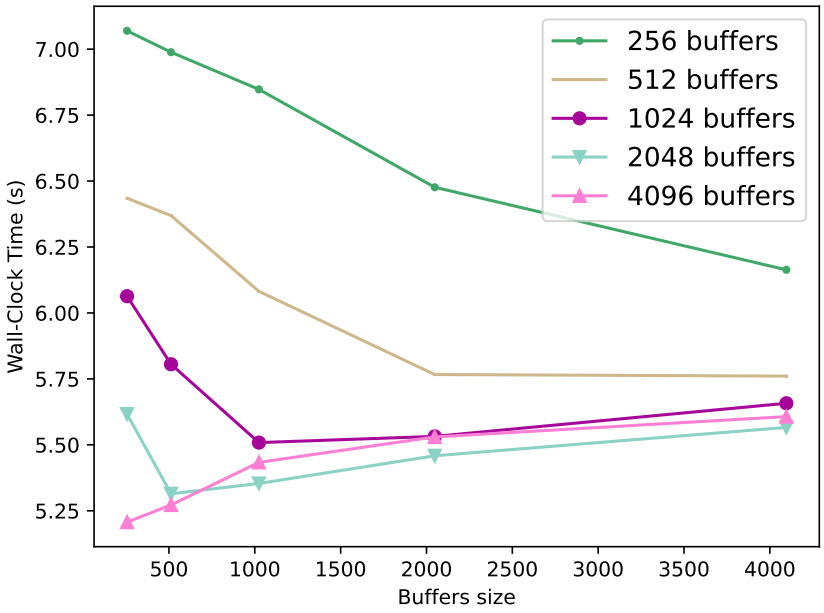
(a) Simulation time of the PARSEC `water_nsquared` with variations of number of buffers and buffer size with **1** vCPUs



(b) Simulation time of the PARSEC `water_nsquared` with variations of number of buffers and buffer size with **2** vCPUs



(c) Simulation time of the PARSEC `water_nsquared` with variations of number of buffers and buffer size with **16** vCPUs



(d) Simulation time of the PARSEC `water_nsquared` with variations of number of buffers and buffer size with **32** vCPUs

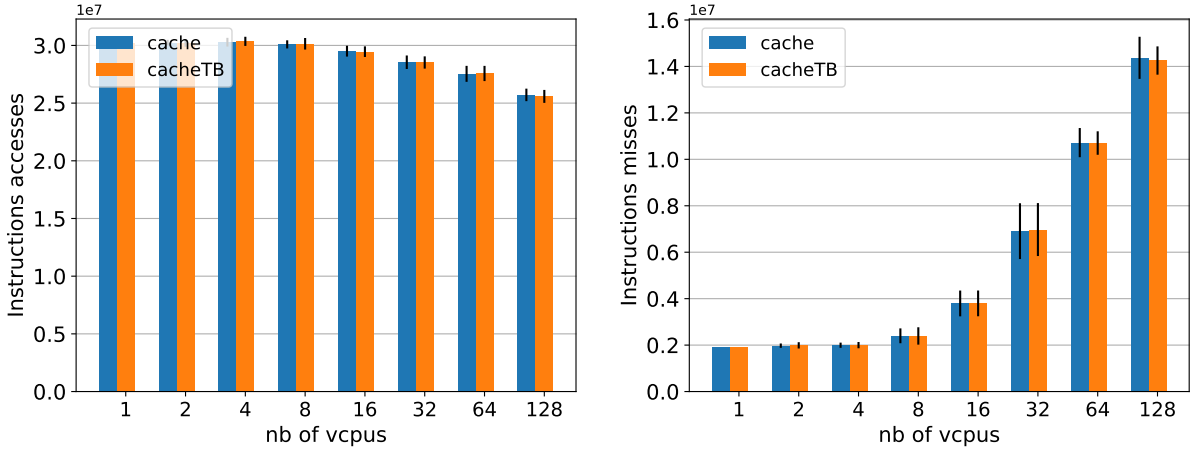
Figure C.1: Simulation time of the PARSEC `water_nsquared` with variations of number of buffers and buffer size with 1,2,16 and 32 vCPUs

Appendix D

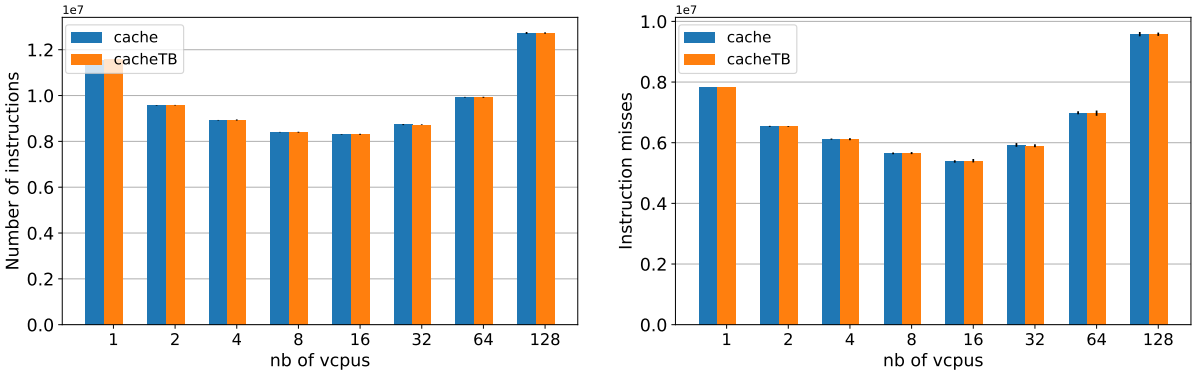
Cache hierarchy per virtual CPU: L1i + L1d + L2 evaluation

This Appendix contains the detailed results on the statistics and execution time for all the benchmarks that are not in Section 6.5.

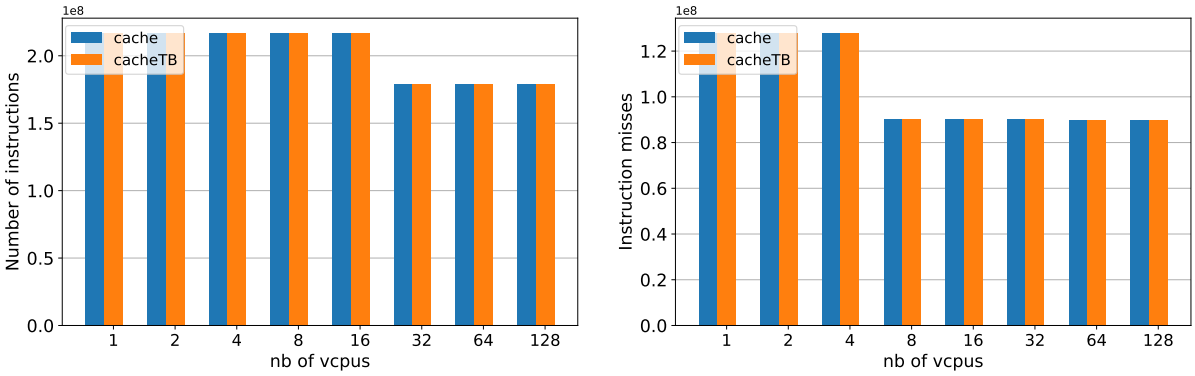
Figures



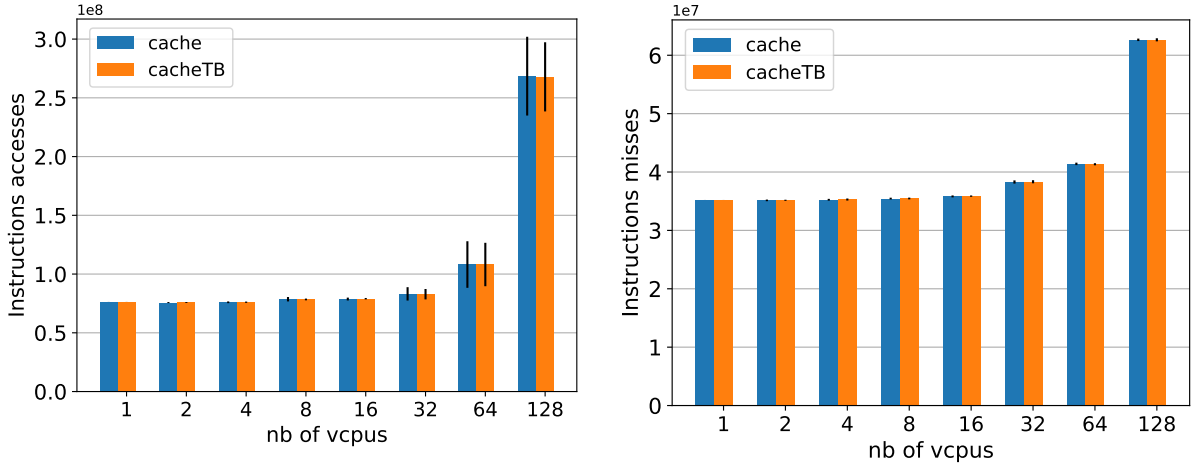
(a) Total **number of instructions** (left) and **misses** (right) for BARNES (log scale on x-axis)



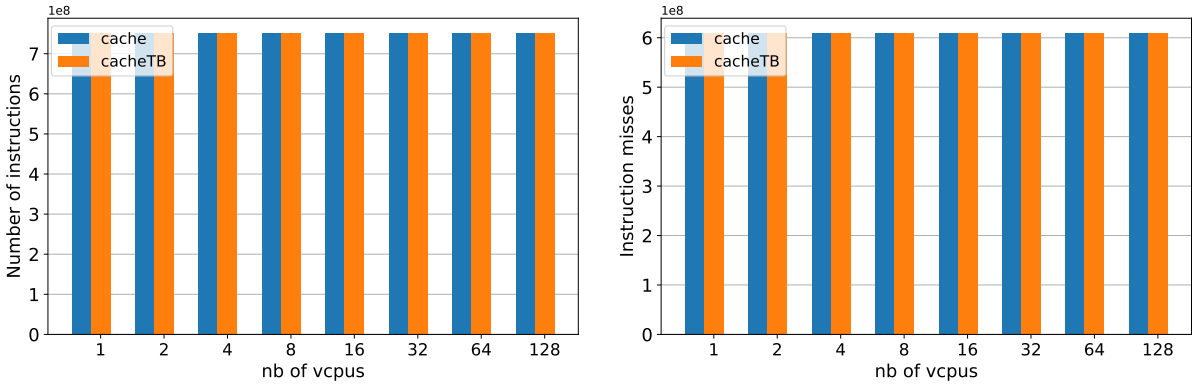
(b) Total **number of instructions** (left) and **misses** (right) for CHOLESKY (log scale on x-axis)



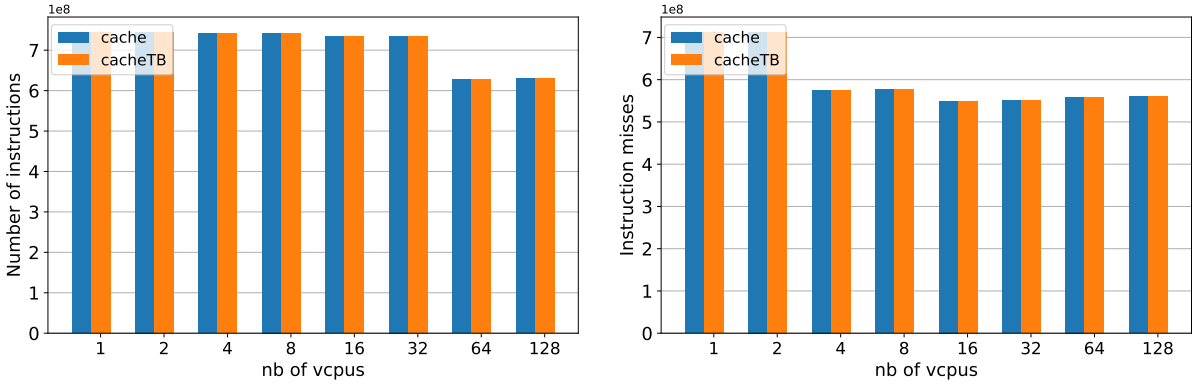
(c) Total **number of instructions** (left) and **misses** (right) for FFT (log scale on x-axis)



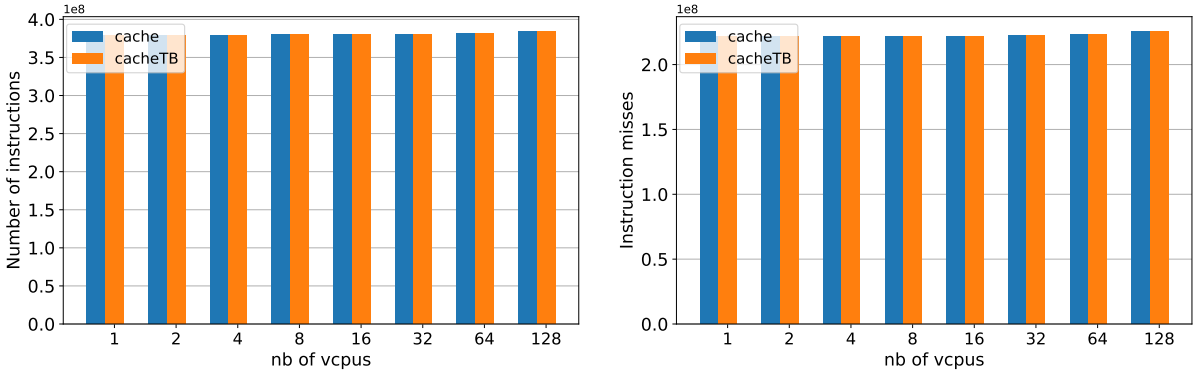
(d) Total **number of instructions** (left) and **misses** (right) for FREQMINE (log scale on x-axis)



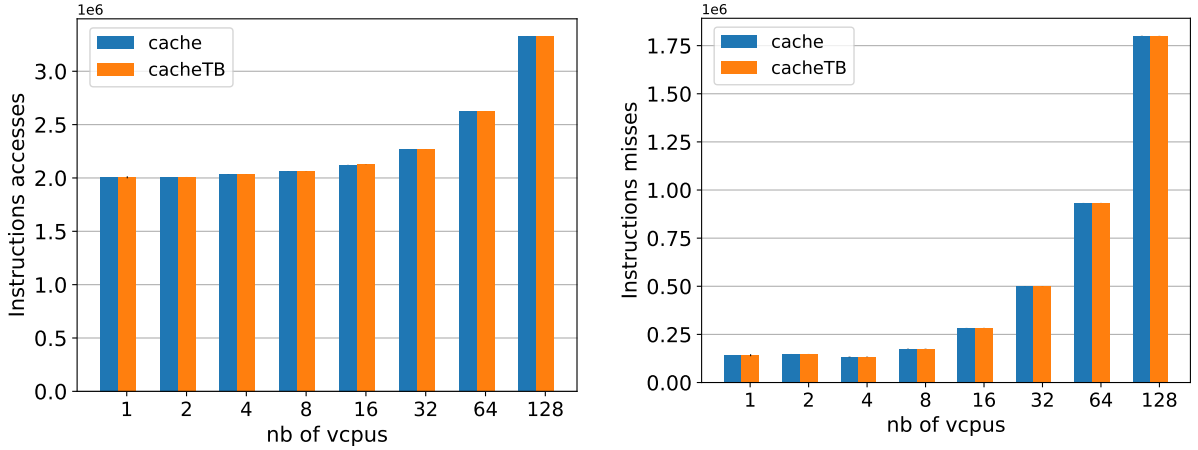
(e) Total **number of instructions** (left) and **misses** (right) for LU_NCB (log scale on x-axis)



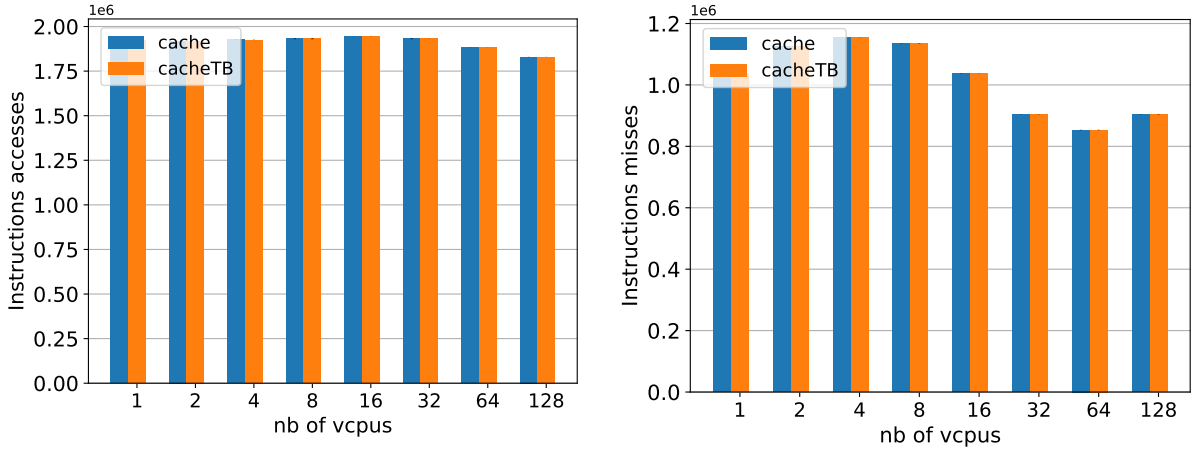
(f) Total **number of instructions** (left) and **misses** (right) for OCEAN_CP (log scale on x-axis)



(g) Total **number of instructions** (left) and **misses** (right) for RADIX (log scale on x-axis)

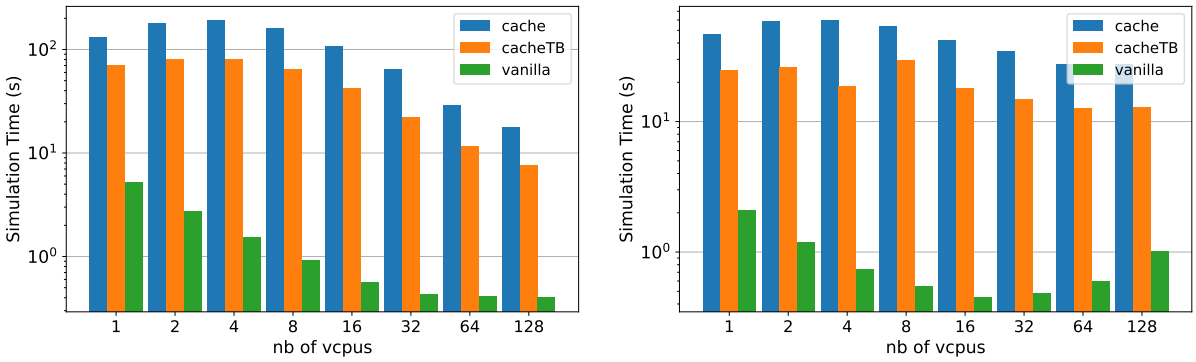


(h) Total **number of instructions** (left) and **misses** (right) for WATER_NSQUARED (log scale on x-axis)

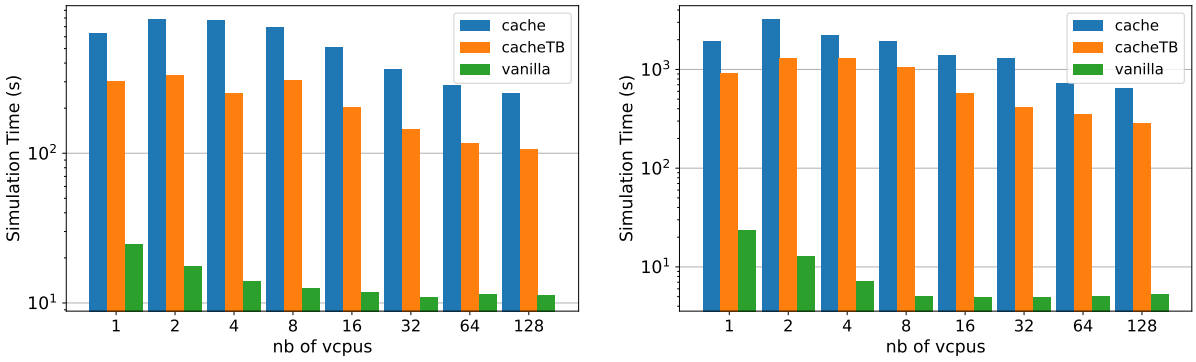


(i) Total **number of instructions** (left) and **misses** (right) for WATER_SPATIAL (log scale on x-axis)

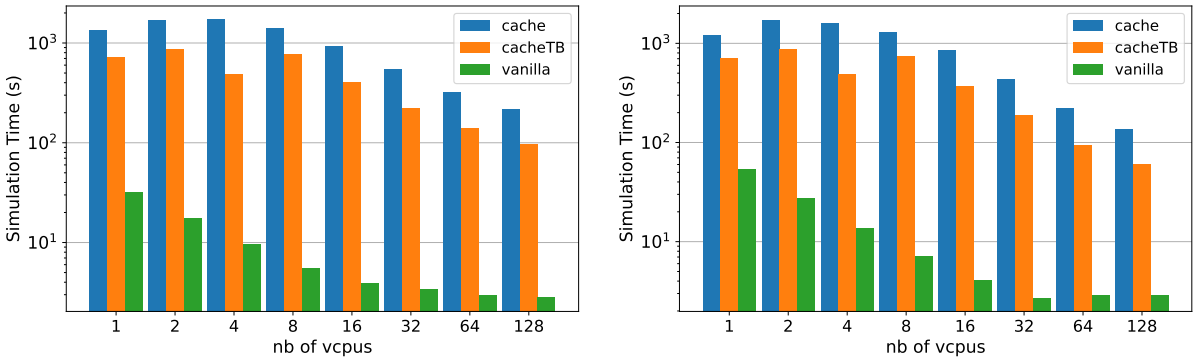
Figure D.1: L2 cache statistics in QEMU `user`-mode of all the other PARSEC



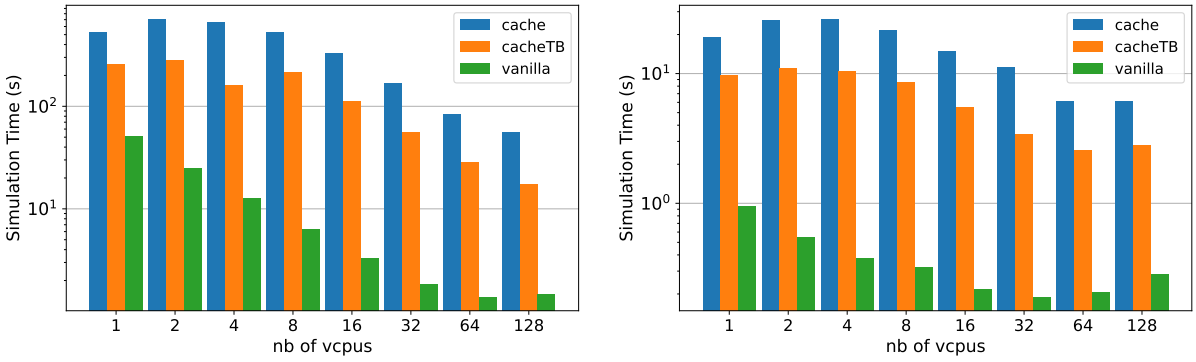
(a) Simulation time of BARNES (left) and CHOLESKY (right) (log-log scale)



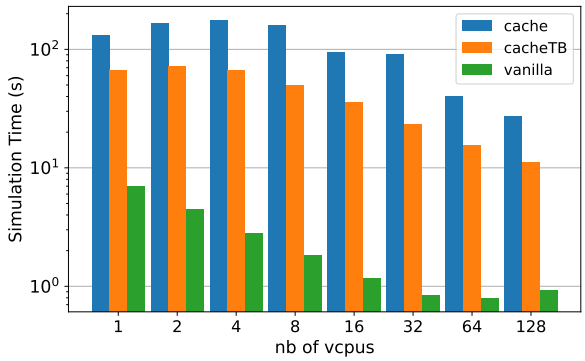
(b) Simulation time of FFT (left) and FREQMINE (right) (log-log scale)



(c) Simulation time of LU_NCB (left) and OCEAN_CP (right) (log-log scale)



(d) Simulation time of RADIX (left) and WATER_NSQUARED (right) (log-log scale)



(e) Simulation time of WATER_SPATIAL (log-log scale)

Figure D.2: Simulation time with L1i + L1d + L2 in QEMU `user`-mode of all the other PARSEC

Publications

International Conferences

– Marie Badaroux and Frédéric Pétrot. ”Arbitrary and Variable Precision Floating-Point Arithmetic Support in Dynamic Binary Translation,” 26th Asia and South Pacific Design Automation Conference (ASP-DAC), Tokyo, Japan, pp. 325-330. https://doi.org/10.1145/3394885.3431416

– Marie Badaroux, Saverio Miroddi and Frédéric Pétrot. ”To Pin or Not to Pin: Asserting the Scalability of QEMU Parallel Implementation”, 24th Euromicro Symposium on Digital Systems Design (DSD), pp. 238-245. https://doi.org/10.1109/DSD53832.2021.00045

Workshops

– Marie Badaroux, Julie Dumas, and Frédéric Pétrot. 2023. Fast Instruction Cache Simulation is Trickier than You Think. In Proceedings of the DroneSE and RAPIDO: System Engineering for constrained embedded systems (RAPIDO ’23). Association for Computing Machinery, pp. 48–53. https://doi.org/10.1145/3579170.3579261

Informal Communications

– Poster presentation of the paper ”Arbitrary and Variable Precision Floating-Point Arithmetic Support in Dynamic Binary Translation”, RISC-V Spring Week, 2022,

– Poster presentation of my PhD contributions, ”Fast and accurate simulation of multi/many-core SoCs”, FETCH Winter School, 2022.

Bibliography

[AAG+16] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. Posix abstractions in modern operating systems: The old, the new, and the missing. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–17, 2016. (Cited on page 54.)

[ABvK+11] Oscar Almer, Igor Böhm, Tobias Edler von Koch, Björn Franke, Stephen Kyle, Volker Seeker, Christopher Thompson, and Nigel Topham. Scalable multi-core simulation using parallel dynamic binary translation. In *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 190–199, 2011. (Cited on pages 10 and 19.)

[Amd67] Gene Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. pages 483–485, April 1967. (Cited on page 38.)

[Bar93] Greg Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 261–270, 1993. (Cited on page 54.)

[BBB+11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, aug 2011. (Cited on pages 9, 26, and 79.)

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46. USENIX, 2005. (Cited on pages 13 and 29.)

[Ben15] Alex Bennée. Towards multi-threaded tcg. In *KVM forum*, 2015. (Cited on pages 20 and 31.)

[Ben20] Alex Bennée. Multi-thread tiny code generator, 2020. https://github.com/qemu/qemu/blob/master/docs/devel/multi-thread-tcg.rst. (Cited on page 20.)

[BFT10] Igor Böhm, Björn Franke, and Nigel Topham. 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*, pages 1–10, 2010. (Cited on page 22.)

[BKP20] Hadi Brais, Rajshekar Kalayappan, and Preeti Panda. A survey of cache simulators. *ACM Computing Surveys (CSUR)*, 53:1–32, 02 2020. (Cited on page 25.)

[BKSL08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, 2008. (Cited on page 37.)

[CBBC17] Emilio G. Cota, Paolo Bonzini, Alex Bennée, and Luca P. Carloni. Cross-isa machine emulation for multicores. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 210–220, 2017. (Cited on page 20.)

[CC19] Emilio Cota and Luca Carloni. 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*, pages 74–87. ACM, 2019. (Cited on page 23.)

[CK94] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. *SIGMETRICS Perform. Eval. Rev.*, 22(1):128–137, may 1994. (Cited on page 23.)

[CM96] Cifuentes and Malhotra. Binary translation: static, dynamic, retargetable? In *1996 Proceedings of International Conference on Software Maintenance*, pages 340–349, 1996. (Cited on page 10.)

[CNZ20] Humberto Carvalho, Geoffrey Nelissen, and Pavel Zaykov. mcqemu: Time-accurate simulation of multi-core platforms using qemu. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 81–88, 2020. (Cited on page 22.)

[CP78] Richard P. Case and Andris Padegs. Architecture of the ibm system/370. *Communications of the ACM*, 21(1):73–96, 1978. (Cited on page 8.)

[CPVM10] Juan Castillo, Hector Posadas, Eugenio Villar, and Marcos Martinez. Fast instruction cache modeling for approximate timed hw/sw co-simulation. In *Proceedings of the 20th Symposium on Great Lakes Symposium on VLSI*, page 191–196. IEEE, 2010. (Cited on page 26.)

[DBK+16] Guillaume Delbergue, Mark Burton, Frederic Konrad, Bertrand Le Gal, and Christophe Jego. Qbox: an industrial solution for virtual platform simulation using qemu and systemc tlm-2.0. In *8th European Congress on Embedded Real Time Software and Systems*, 2016. (Cited on page 20.)

[DCHC11] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. Pqemu: A parallel system emulator based on qemu. In *1st International QEMU Users' Forum*, pages 35–38, 2011. (Cited on page 20.)

[dGGL16] Amanieu d'Antras, Cosmin Gorgovan, Jim Garside, and Mikel Luján. Optimizing indirect branches in dynamic binary translators. *ACM Trans. Archit. Code Optim.*, 13(1), apr 2016. (Cited on page 21.)

[DGVS14] Luis Díaz, Eduardo González, Eugenio Villar, and Pablo Sánchez. Vippe, parallel simulation and performance analysis of multi-core embedded systems on multi-core platforms. In *Design of Circuits and Integrated Systems*, pages 1–7, 2014. (Cited on page 19.)

[DM10] Arnaldo Carvalho De Melo. The new linux "perf" tools. In *Linux Kongress*, volume 18, 2010. http://vger.kernel.org/~acme/perf/lk2010-perf-acme.pdf. (Cited on page 69.)

[EH98] Jan Edler and Mark D Hill. Dinero iv trace-driven uniprocessor cache simulator, 1998. https://pages.cs.wisc.edu/~markhill/DineroIV/. (Cited on pages 8 and 25.)

[Far18] Antoine Faravelon. *Acceleration of memory accesses in dynamic binary translation*. PhD thesis, 10 2018. (Cited on page 21.)

[FGP21] Antoine Faravelon, Olivier Gruber, and Frédéric Pétrot. Removing load/store helpers in dynamic binary translation. In *Multi-Processor System-on-Chip 1*, chapter 7, pages 133–160. John Wiley & Sons, Ltd, 2021. (Cited on page 21.)

[FKMM15] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. Beyond processor-centric operating systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015. (Cited on page 6.)

[Gar14] Timothy Garnett. Dynamic optimization of ia32 applications under dynamorio. 05 2014. (Cited on page 21.)

[GDBAS20] Davood Ghatrehsamani, Chavit Denninnart, Josef Bacik, and Mohsen Amini Salehi. The art of cpu-pinning: Evaluating and improving the performance of virtualization and containerization platforms. In *Proceedings of the 49th International Conference on Parallel Processing*, ICPP '20, New York, NY, USA, 2020. Association for Computing Machinery. (Cited on page 73.)

[GFKW11] Pawel Gepner, David L. Fraser, Michal Filip Kowalik, and Kazimierz Waćkowski. Evaluating new architectural features of the intel (r) xeon (r) 7500 processor for hpc workloads. *Computer Science*, 12:5–17, 2011. (Cited on page 40.)

[GFP09] Marius Gligor, Nicolas Fournel, and Frédéric Pétrot. 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*, pages 71–80. ACM, 2009. (Cited on page 26.)

[Gil07] Damien Gille. Study of different cache line replacement algorithms in embedded systems. Master's thesis, KTH—Royal Institute of Technology, Stockholm, 2007. (Cited on page 50.)

[Gre13] Brendan Gregg. Thinking methodically about performance. *Communications of the ACM*, 56(2):45–51, 2013. (Cited on page 69.)

[Gui11] Christophe Guillon. Program instrumentation with qemu. In W. Mueller and F. Pétrot, editors, *1st International QEMU Users' Forum*, volume 1, pages 15–18, 2011. (Cited on page 23.)

[Ham15] Julian Hammer. pycachesim, 2015. https://github.com/RRZE-HPC/pycachesim. (Cited on page 25.)

[HDBZ15] Byron Hawkins, Brian Demsky, Derek Bruening, and Qin Zhao. Optimizing binary translation of dynamically generated code. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, page 68–78, USA, 2015. IEEE Computer Society. (Cited on page 21.)

[Her93] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993. (Cited on page 54.)

[HHY+12] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. Hqemu: A multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, page 104–113, New York, NY, USA, 2012. Association for Computing Machinery. (Cited on pages 21 and 23.)

[HM93] Maurice Herlihy and J Eliot B Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, 1993. (Cited on page 54.)

[JC17] Michael J. Clark. busybear-linux, 2017. https://github.com/michaeljclark/busybear-linux. (Cited on page 40.)

[JMW+22] Fatma Jebali, Oumaima Matoussi, Arief Wicaksana, Amir Charif, and Lilia Zaourar. 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*, page 47–52. ACM, 2022. (Cited on page 25.)

[KSC+20] Martin Kristien, Tom Spink, Brian Campbell, Susmit Sarkar, Ian Stark, Björn Franke, Igor Böhm, and Nigel Topham. Fast and correct load-link/store-conditional instruction handling in dbt systems. In *CASES'20: Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2020. (Cited on page 19.)

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005. (Cited on pages 13 and 23.)

[LHW+14] Yi-Hong Lyu, Ding-Yong Hong, Tai-Yi Wu, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. Dbill: An efficient and retargetable dynamic binary instrumentation framework using llvm backend. *Acm Sigplan Notices*, 49(7):141–152, 2014. (Cited on page 23.)

[Mag97] Peter S. Magnusson. Efficient instruction cache simulation and execution profiling with a threaded-code interpreter. In *Proceedings of the 29th conference on Winter simulation*, pages 1093–1100. IEEE, 1997. (Cited on page 26.)

[Man21] Mahmoud Mandour. Cache modelling tcg plugin, 2021. https://www.qemu.org/2021/08/19/tcg-cache-modelling-plugin/. (Cited on page 26.)

[MCDK21] Kévin Mambu, Henri-Pierre Charles, Julie Dumas, and Maha Kooli. Instruction set design methodology for in-memory computing through qemu-based system emulator. In *2021 IEEE International Workshop on Rapid System Prototyping (RSP)*, pages 43–49. IEEE, 2021. (Cited on page 24.)

[Mic14] Luc Michel. *Contributions to dynamic binary translation : instruction parallelism support and optimized translators generator*. PhD thesis, 12 2014. (Cited on page 11.)

[Mir21] Saverio Miroddi. Qemu-pinning, 2021. https://github.com/64kramsystem/qemu-pinning. (Cited on page 29.)

[MTB11] Abdelhafid Mazouz, Sid-Ahmed-Ali Touati, and Denis Barthou. Performance evaluation and analysis of thread pinning strategies on multi-core platforms: Case study of spec omp applications on intel architectures. In *2011 International Conference on High Performance Computing & Simulation*, pages 273–279, 2011. (Cited on page 33.)

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007. (Cited on page 23.)

[NS15] Alejandro Nicolas and Pablo Sanchez. Parallel native-simulation for multi-processing embedded systems. In *2015 Euromicro Conference on Digital System Design*, pages 543–546, 2015. (Cited on page 19.)

[Oda23] Akihiko Odaki. [qemu] plugins: Allow to read registers, 2023. https://patchew.org/QEMU/20231019102657.129512-1-akihiko.odaki@daynix.com. (Cited on page 25.)

[OYTN17] Katsumi Okuda, Minoru Yoshida, Haruhiko Takeyama, and Minoru Nakamura. Automated generation of dynamic binary translators for instruction set simulation. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 214–219, 2017. (Cited on page 18.)

[PBC+15] Andrej Podzimek, Lubomír Bulej, Lydia Y. Chen, Walter Binder, and Petr Tuma. Analyzing the impact of cpu pinning and partial cpu loads on performance and energy efficiency. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 1–10. IEEE, 2015. (Cited on page 73.)

[PY15] Louis-Noël Pouchet and Tomofumi Yuki. Polybench/c 4.1, 2015. http://polybench.sourceforge.net. (Cited on page 59.)

[RCBJ11] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, 2011. (Cited on page 9.)

[RRD17] Simon Rokicki, Erven Rohou, and Steven Derrien. Hardware-accelerated dynamic binary translation. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1062–1067, 2017. (Cited on page 21.)

[RSR16] Alvise Rigo, Alexander Spyridakis, and Daniel Raho. Atomic instruction translation towards a multi-threaded qemu. In *30th European Conference on Modelling and Simulation*, pages 1–9, 2016. (Cited on page 20.)

[SE04] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. *SIGPLAN Not.*, 39(4):528–539, apr 2004. (Cited on page 23.)

[SHP12] Hao Shen, Mian-Muhammad Hamayun, and Frédéric Pétrot. Native simulation of mpsoc using hardware-assisted virtualization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(7):1074–1087, 2012. (Cited on page 19.)

[SR15] Gabriel Southern and Jose Renau. Deconstructing parsec scalability. In *Proc. of the Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2015. (Cited on pages 38 and 62.)

[SWF20] Tom Spink, Harry Wagstaff, and Björn Franke. A retargetable system-level dbt hypervisor. *ACM Trans. Comput. Syst.*, 36(4), may 2020. (Cited on page 18.)

[UC00] David Ung and Cristina Cifuentes. Machine-adaptable dynamic binary translation. *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, 35, 04 2000. (Cited on page 18.)

[VDTT14] Tran Van Dung, Ittetsu Taniguchi, and Hiroyuki Tomiyama. Cache simulation for instruction set simulator qemu. In *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, pages 441–446. IEEE, 2014. (Cited on page 26.)

[VMK22] Mihir Vora, Priyabarata Mishra, and Anirudha Shrikant Kurhade. Integration of pycachesim with qemu. In *2021 4th International Conference on Recent Trends in Computer Science and Technology (ICRTCST)*, pages 27–30, 2022. (Cited on page 25.)

[Wei08] Josef Weidendorfer. Sequential performance analysis with callgrind and kcachegrind. In *Tools for High Performance Computing*, pages 93–113. Springer, 2008. (Cited on page 25.)

[WHK+07] Cheng Wang, Shiliang Hu, Ho-seop Kim, Sreekumar R. Nair, Mauricio Breternitz, Zhiwei Ying, and Youfeng Wu. Stardbt: An efficient multi-platform dynamic binary translation system. In *Proceedings of the 12th Asia-Pacific Conference on Advances in Computer Systems Architecture*, ACSAC'07, page 4–15, Berlin, Heidelberg, 2007. Springer-Verlag. (Cited on page 18.)

[WLC+11] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. Coremu: a scalable and portable parallel full-system emulator. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 213–222, 2011. (Cited on page 20.)

[WM08] Vincent M. Weaver and Sally A. McKee. Are cycle accurate simulations a waste of time. In *Proc. 7th Workshop on Duplicating, Deconstructing, and Debunking*, pages 40–53. Citeseer, 2008. (Cited on page 9.)

[WR96] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '96, page 68–79, New York, NY, USA, 1996. Association for Computing Machinery. (Cited on page 18.)