Full-system After-cache Memory Tracing for Multi-core Systems using a Distributed Cache Simulator

Dorian de Koning

Full-system After-cache Memory Tracing for Multi-core Systems using a Distributed Cache Simulator

Master's Thesis in Computer Science

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Dorian de Koning

March 13th 2020

Author
Dorian de Koning

Title
Full-system After-cache Memory Tracing for Multi-core Systems using a Distributed Cache Simulator

MSc presentation
March 24th 2020

Graduation Committee

Dr. ir. J. A. Pouwelse	Delft University of Technology
Dr. J. S. Rellermeyer	Delft University of Technology
Dr. ir. Z. Al-Ars	Delft University of Technology

Abstract

The gap between CPU and memory performance becomes increasingly larger. Together with a growing memory pressure caused by higher CPU core counts combined with multi-tenant systems, this causes the need for new memory technologies. Recently, various technologies are becoming available for commercial use. Examples of these technologies are memory types like non-volatile RAM. These technologies generally have different characteristics than traditional DRAM. To be able to fully utilise the potential of these new memory types, a better understanding of memory usage in modern systems is required. A way to gain a better understanding is through memory traces.

Solutions that are currently available either do not support multi-core architectures or cause a severe slowdown. Therefore, this thesis presents a novel approach to gather full-system after-cache memory access traces. The proposed system is a hybrid framework which consists of the QEMU emulator combined with a custom distributed cache and page table simulator. A modified version of QEMU, called QMEMU, is devised to improve tracing performance and allow tracing instruction fetches. By leveraging the existing tracing functionality of QEMU only a small amount of modifications have to be made to QEMU. The traces produced by QMEMU contain virtual addresses. However, for accurate cache simulation, the physical addresses have to be used. Tracing the physical address instead of the virtual address for each memory access is shown to cause a 70% slowdown when using QMEMU.

To find these physical addresses for the traced accesses, a novel approach is employed. This approach simulates the guest page tables outside the critical path for memory tracing and therefore does not decrease performance. Using QMEMU traces can be gathered with a speedup of up to 42.6 times over the gem5 simulator for benchmarks of the PARSEC suite.

In the second part of the framework, which performs memory, cache, and page table simulation, cache simulation is found the most computationally intensive task. Therefore in the proposed framework cache simulation is performed in a parallel and distributed manner. Most modern systems use set-associative caches, simulation can be parallelised without reducing accuracy by dividing the memory access traces based on these cache sets. Using this approach 10 Million accesses can be processed per second by the simulator when simulating a single modern cache hierarchy. When simulating 7 different cache hierarchies concurrently a throughput of 6 Million accesses per second is reached. The simulated guest page tables provide additional information like the number of accesses or virtual memory size for each process of the guest workload. This information can be used to decrease the size of the semantic gap between memory traces and their meaning. The proposed framework is evaluated by comparing it to CMP$im and gem5 using the PARSEC benchmark suite.

Preface

This thesis is the concluding part of work for the Master's degree in Computer Science at the Delft University of Technology.

First and foremost, I want to thank my supervisor Dr. J.S. Rellermeyer for the invaluable guidance during the stages of developing this thesis. Furthermore, I would like to express my gratitude to Dr .ir. J.A. Pouwelse and Dr. ir. Al-Ars for being part of the graduation committee.

Finally, I also want to thank my friends and family. Especially, my parents for their support during the journey starting in primary school and ending with this thesis and in special during the last months. Without their endless support, I would not have been able to complete my studies successfully.

Dorian de Koning

Nootdorp, The Netherlands
March 13th 2020

Contents

Chapter 1

Introduction

The popularity of infrastructure as a service (IaaS) cloud systems has tremendously increased over the last years and is expected to increase even more [2]. One of the key concepts behind these cloud systems is multi-tenancy. In these multi-tenant systems multiple applications, often from separate users, run on the same hardware each in a separate virtual machine (VM) or container. Many of these show low activity and are only occasionally active [43]. Resources such as CPU or network bandwidth can be shared using advanced scheduling mechanics like Multiqueue or Flow Scheduling [32]. These techniques allow more containers to be scheduled on the same machine. Sharing memory, however, is a more difficult problem. Techniques like virtual memory allow the system to be over-allocated by moving unused memory pages to the disk. However, when an inactive VM becomes active, its pages stored on the relatively slow disk first have to be moved into memory before it continues. Paging can thus cause a significant increase in the latency of these containers or VMs [35].

Changes in how systems use modern multi-core architectures with higher core count further increase the memory pressure resulting in an increasing need for systems with larger memory capacity. A well-known law in computer science, Moore's law, states that the number of transistors in a CPU doubles roughly every two years. This law has held up for over more than 40 years. Unfortunately, there is an extension of this law which states that the access time to memory improves at less than 10% per year. Because of this difference, the gap between CPU and memory performance grows with roughly 50% each year [21]. Apart from the gap in performance the difference in energy usage in modern systems also increases. Energy-proportionality is a metric used to quantify the energy usage of a system relative to the amount of useful work that is done; this metric is proposed by Barroso et al. [15]. The increasing gap between memory and CPU causes the energy usage of memory to be one of the least energy-proportional components of modern systems.

A recent trend in computer memory is a new type of memory, non-volatile RAM (NVRAM). This type of memory generally provides higher capacity and band-

width at a lower cost than traditional DRAM. Moreover, these types of memory typically provide orders of magnitude better latencies than solid-state drives. DRAM continuously requires energy to maintain its state, whereas non-volatile counterparts can maintain their state without consuming energy. Compared to DRAM, NVRAM has several disadvantages, it generally requires more energy to read and write data [30], has a significant difference between read and write speed, and it has a limited lifetime. Additionally, only complete blocks of memory can be read or written instead of the byte-level granularity available for DRAM [34]. Together these properties make NVRAM unsuitable to replace DRAM completely.

Instead of using NVRAM as the main memory of a system, it could be used as a very high-speed hard disk. This way, it could be used as a high-speed swap disk which unfortunately does not allow the stored data to be directly read or modified. In current operating system paging implementations, a page fault occurs when a program tries to access a page not currently residing in main memory. On a page fault, the corresponding page is moved entirely into DRAM, which will increase latency. There is no consensus on the best way to use these new types of memory within a system yet.

Because of these different properties of arising memory technologies like NVRAM, it is essential to gain a better understanding of how modern systems use their memory. This knowledge can then be used to design, implement and evaluate new memory allocation techniques to use NVRAM and future memory technologies as efficiently as possible.

1.1 Problem statement

As the availability of new non-volatile memory technologies increases, new memory management policies must utilise this potential. Devising new memory management strategies requires a thorough understanding of the memory requirements of different applications and also demands ways to evaluate these new strategies. Because the operating system kernel performs most of the memory management logic, full-system memory tracing is preferred over collecting traces for single binary applications. Currently, full-system memory tracing is only available in systems like MARSSx86 [41], gem5 [19] and Simutrace [44] at the cost of a substantial increase in runtime. The MARSSx86 simulator is approximately 1000 times slower than a high-end machine [42]. It reaches a simulation speed of 200.000 instructions per second [6] for workloads in the PARSEC benchmark suite [17]. A modern CPU running at 2Ghz like the one simulated by MARSSx86 can execute up to 2 billion instructions per second per core, assuming one cycle per instruction [5]. Memory access traces can thus currently only be collected for small, non-realistic, scenarios, specific hardware platforms, or single binary applications. Because the main goal for gathering traces is to gain more insight into the usage of the main memory, memory accesses which hit the cache are not of interest.

Following from this problem statement the primary goal of this thesis is to pro-

pose a framework for gathering full-system memory traces for realistic workloads. This thesis revolves around the following research question:

How can full-system after-cache memory access traces be gathered?

This question naturally lends itself to be divided in four subquestions:

1. *How can full-system memory access traces be gathered with minimal performance impact?*
   One of the problems with available approaches for tracing memory accesses is the fact that their slowdown makes it infeasible to trace realistic workloads. Therefore one of the subquestions in this thesis is how to efficiently trace all memory accesses performed by a system in a way that has a minimal performance impact.

2. *How can multi-core CPU caches be simulated efficiently to convert before-cache traces to after-cache traces?*
   Following the main research question of this thesis, the main goal for which traces are gathered is evaluating the use of main memory of a system. All modern CPUs contain caches to reduce the number of memory accesses that have to be handled by the relatively slow main memory of a computer. However, when developing a system to gain more insight in memory usage, the accesses which are handled by the cache are thus not useful. Therefore the memory accesses performed by the CPU have to be filtered such that only cache accesses which miss the CPU cache and are handled by the main memory of a system remain.

3. *How can different types of CPU caches be simulated in parallel?*
   To be able to evaluate the behaviour of different cache hierarchies on a workload, multiple caches have to be simulated. Because the simulation of different cache hierarchies is independent of each other, they could be simulated in parallel instead of sequentially, which could decrease the total runtime of the simulator when simulating complex caches. From this problem the question arises how this parallelisation can be done, ideally without compromising accuracy.

4. *How can page tables be traced to reduce the semantic gap?*
   A problem that arises when analysing traces containing only memory accesses is the lack of semantic information about processes and virtual addresses of the workload [31]. This gap between memory traces and their meaning is called the semantic gap. The state of the control registers and page tables in a system contains valuable information about the memory usage of operating systems and the processes running on it. Therefore, tracing their values could be useful to help close this semantic gap.

1.2 Research approach and contributions

To answer the main research question and the four sub-questions, different techniques are investigated. This is done by designing and implementing a framework based on existing techniques and newly devised methods. Where possible, design decisions are made based on experimental results. Finally, to validate the design, the framework will be implemented and compared to other state of the art solutions.

By answering the posed research questions, this thesis contributes a novel approach to full-system memory tracing. The framework can create full-system after-cache traces for real-world workloads for unmodified hosts, no known available solution is able to do this. The implementation of the framework focusses on the x86, but the concept of the framework is independent of operating system or architecture.

Chapter 2

Overview

Designing systems that leverage the full potential of NVRAM requires a thorough understanding of memory access patterns of real-world applications. Therefore, it is crucial to have information on the memory usage of different configurations and workloads. Memory access traces can be used to gain more information about the memory usage of a system and contain information on every memory access performed by the system. Each traced event in such a trace contains the address accessed, whether the system read or wrote to the memory location and the clock tick off the access. Generally, these memory traces can be divided into two categories. The first category contains access traces of memory accesses performed by a single program, so-called single-binary traces. The other contains traces of all memory accesses in the system, including those performed by the operating system kernel. Currently, no traces of multi-core real-world workloads are publicly available. However, there is a wide variety of tools available for generating multi-core full-system memory traces. These, however, are not able to generate after-cache traces or causes a significant performance impact making it inviable to create full-system traces for real-world workloads. Another advantage of being able to generate these traces instead of using publicly accessible traces is the fact that these tools can be used to evaluate and analyse the behaviour of newly devised systems that use new memory technologies and compare them to traces of conventional systems.

2.1 Related work

2.1.1 Hardware devices

For creating memory access traces hardware tracing devices are available. These devices are used by placing a physical tap on one of the buses in a system. These devices attach to a bus and receive all messages which are sent on that bus. Hardware tracing devices do generally not influence the performance of the system. Devices such as BACH [28] are able to monitor all traffic on a bus. The buses these devices typically get connected to can have a bandwidth of up to multiple

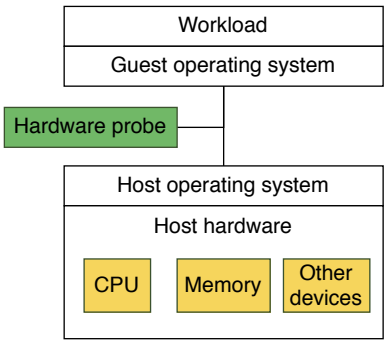


Figure 2.1: Hardware device based tracing, the location where the memory accesses are traced is green

Gigabytes per second for systems with multiple CPUs [14]. Typically hardware tracing devices have a buffer, but if trace events are produced faster than they can be processed over a longer period of time, the buffer fills. When the trace buffer is full, no more trace events can get captured. Therefore, only traces of small, relatively short workloads can be gathered. An alternative is to use complicated storage setups to handle the stream of accesses created by these devices [14]. Furthermore, these solutions require physical access to the platforms under investigation. Therefore they do not allow modifying parameters like CPU core count or cache size as easily as they can be modified when using simulation-based tracing. Additionally, most of these approaches use expensive hardware like logic analysers [14].

2.1.2 Sampling

Sampling of memory accesses is tracing a subset of all accesses that occur in a system. One of the tools that can be used to sample memory accesses is perf [11], a tool for analysing the performance of the entire system in Linux. Another tool available is VTune Profiler developed by Intel [8]. Both VTune and perf support sampling of memory accesses using hardware performance counters. However, both solutions can only sample a subset of all memory accesses and only support memory access sampling for modern Intel CPUs [8]. Because ideally memory tracing should capture all accesses and work independent of the operating system, CPU vendor or even processor architecture, these and similar solutions are not suitable.

2.1.3 Binary instrumentation

For gathering traces of a specific workload, a technique called binary instrumentation can be used. In binary instrumentation, a program is used which modifies a binary running in a system by inserting additional instructions. Specific instructions can be inserted which instrument each memory access. Examples of binary

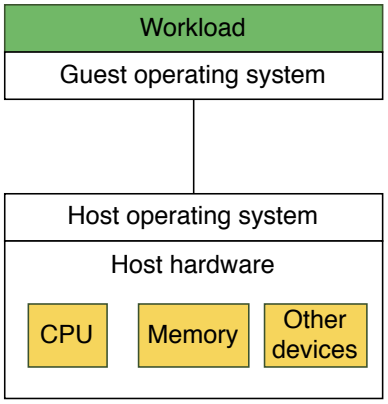
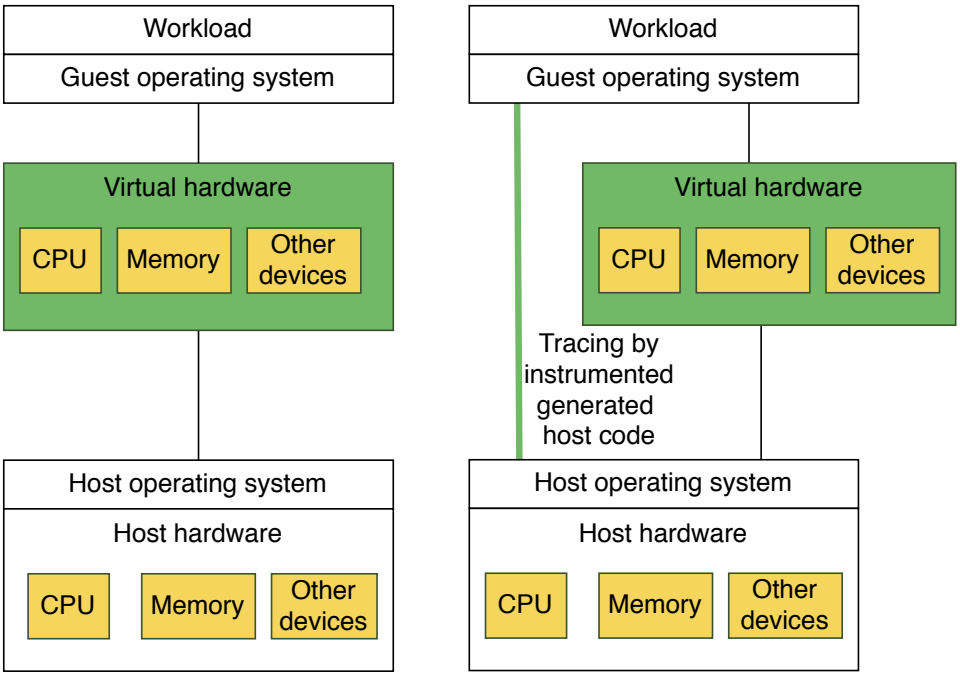


Figure 2.2: Binary instrumentation based tracing, the location where the memory accesses are traced is green

instrumentation tools are Pin by Intel [36] and Valgrind [39]. Binary instrumentation using tools like these only allow access tracing for single binary applications, not for complete operating systems.

Another technique to gather memory access traces is to simulate the complete system under test. There are three main categories of system simulators. Simulators that only simulate the system calls the guest makes, full-system simulators and cycle-accurate simulators. Full-system simulators purely aim to stimulate or translate the instructions executed by the guest to host instructions whereas cycle-accurate simulators aim to simulate the complete hardware as accurately as possible. User mode emulators, like QEMU [16], translate the instructions in a binary compiled for the emulated guest to host instructions and emulate the system calls a guest binary makes. This approach allows binaries compiled for a different architecture to be executed on a host of a different instruction set architecture. In these simulators, only instructions executed by the guest binary are emulated, but the memory accesses performed by the kernel in system calls are not. Therefore the memory accessed by the kernel in the simulated system calls or kernel processes can not be traced.
The second type of simulators is full-system simulators which simulate devices like memory, hard drives and network interface devices. Generally, any operating system kernel can be configured to run on these types of simulators. Oracle VirtualBox [40] is an example of a full-system simulator, besides the previously described user mode, another operating mode of QEMU is the full-system mode. When QEMU runs in full-system mode, all CPU instructions executed by a guest operating system are simulated and thus also the instructions executed by all user processes running on the operating system. Full-system simulators also simulate the behaviour of peripheral devices instead of using the host peripheral devices. Generally, these simulators employ a large number of optimisation techniques to be able to simulate the guest at near-native speed.

Modern processors even have hardware support for virtualisation in the form of virtualisation extensions adding various virtualisation specific instructions to the instruction set. These instructions can be used to enter and exit a virtual execution mode. When running in this mode, the host operating system is protected, but the guest is able to run with full privileges. This allows instructions of a guest with the same instruction set architecture (ISA) as the host to be directly executed on the host processor. When the guest has a different ISA than the host, guest instructions have to be translated. Another optimisation found in modern processors is hardware support for simulating the guest memory management unit (MMU). Instead of simulating the MMU hardware which has a significant overhead on every memory access, the address translation from virtual to physical addresses uses the hardware of the host machine. This is done using shadow page tables where the host sets up a page table mapping from guest virtual addresses to host physical addresses in the host page tables to be used by the guest



(a) Cycle accurate based simulation (b) QEMU based simulation

Figure 2.3: Comparison between cycle-accurate simulation and QEMU based tracing, the location where memory accesses are traced is marked in green

Cycle-accurate simulators are a subset of full-system simulators. An example of a cycle-accurate simulator is gem5 [19]. Cycle-accurate simulators generally simulate a machine on hardware level including the CPU and buses. These simulators are mainly designed for research and design of new architectures. Therefore, these

simulators aim to be as accurate as possible. However, because of their accuracy, combined with the fact that they can not leverage underlying hardware virtualisation support, they are complex and thus generally have a significantly higher runtime than user mode or full-system simulators. Because of their complexity running these simulators can have a significant slowdown. The most widely used gem5 simulator can simulate multi-core CPUs but is not able to use multiple cores of the host to do this simulation. Therefore, simulating systems with multiple cores increases the slowdown even further. An advantage of these simulators is that sometimes memory tracing is already built-in. Regardless, this advantage does not outweigh the orders of magnitude slowdown compared to other full-system simulators.

Out the three types of simulators which where discussed, both cycle-accurate and full-system simulators can be used for full-system memory tracing but full-system simulators are preferred because of lower performance penalty.

QEMU

One of the most widely used full-system simulators is the open-source simulator QEMU. QEMU has multiple different operating modes. When simulating an x86 guest on an x86 system, QEMU can use the kernel-based virtual machine (KVM) to simulate the host target. KVM turns Linux into a hypervisor that allows a host machine to simulate multiple guests. These virtual machines run directly on the hardware and are managed using the virtual machine manager. Guests running on KVM can reach near-native performance. The host CPU can be configured to generate an interrupt for each memory access performed when guest instructions are being executed. Unfortunately, this will have a significant performance impact.

Apart from using KVM, QEMU can also be configured to use the Tiny Code Generator (TCG). In this mode guest operations are translated into TCG operations by the frontend. The backend translates these TCG operations into host instructions. With this intermediate step of TCG instructions QEMU decouples frontends from backends, this allows any combination of frontend and backend to be made. Instructions can be divided into basic blocks which are separated by jump instructions. Instead of translating each instruction individually, translation can be performed block by block. Translated blocks can be cached to prevent translating the same sequence of instruction many times.

This approach taken by QEMU makes it suitable for memory tracing since the backend can be modified to execute a specific sequence of instructions for every guest memory access. Instructions can be instrumented to keep track of which memory addresses are accessed. The additional instructions that have to be executed by the host system to trace memory access will inevitably cause a performance penalty. QEMU and other full-system simulators do not aim to simulate actual hardware as accurately as possible. Therefore CPU caches of the guest are not simulated. Thus when using QEMU, only memory access traces from before the CPU caches can be gathered. However, to answer the primary research question traces

of the memory accesses which missed the cache and are redirected to main memory are needed. Therefore a separate cache simulator can be used to make after-cache traces from the before cache traces QEMU outputs similar to the approach taken in the MARSSx86 simulator [41].
In the MARSSx86 simulator, a hybrid approach is used which combines QEMU and PTLsim [47]. QEMU is used to simulate peripheral devices like hard disks and network interface devices. A custom simulation engine using the cycle-accurate PTLsim to simulate an x86 CPU is used to simulate the CPU and memory devices. Because active development of MARSSx86 has stopped it uses an outdated version of QEMU extended and therefore out of the box it only runs versions 3.2 and lower of the Linux kernel [7]. In addition to this, it does not have support playing back captured traces and is only able to simulate x86 CPUs.

2.1.4 Cache simulation

To translate before-cache memory traces to after-cache traces a cache simulator is used. Various cache simulators are available. Dinero IV [1] is a widely used cache simulator, which can simulate multi-level caches. Dinero IV has relatively high performance; however, it does not support multi-core workloads and cache coherency protocols. Another widely used cache simulator is CMP$im [33] based on the Intel pin framework [36]. Because of its dependence on the binary instrumentation tool Pin, it is only able to simulate single binary userspace workloads instead of a full-system as required. The FM-SIM [37] is also based on Pin and thus has the same problem. A very high-performance multi-core simulator is Sniper [29]. However, it is meant to analyse the performance of the system and is therefore based on an interval simulation model. Interval simulation simulates only certain intervals of the execution and using the prediction of miss events like TLB misses and branch miss predictions, and therefore it does not simulate all memory accesses, but only samples a specific subset of the accesses.

An approach for a distributed cache simulator is Shaman [38]. The proposed simulator divides the simulation process into two parts. The simulator consists of multiple frontend nodes and a single backend node. Each frontend node simulates the instructions executed on a single processor and generates shared memory references. These memory accesses are filtered using reference filtering, where only memory references that may result in a cache miss are passed to the backend. Based on these references, the backend simulates the physical accesses and creates an after-cache memory access trace. Unfortunately, using this reference filtering technique, write-trough and write-back caches cannot be filtered effectively and cause the performance of the overall system to decline significantly. A similar approach is taken in the PriME simulator [26] where instruction simulation is distributed over multiple nodes in a cluster, an average speed of 20 MIPS is achieved at an average simulated running time error of 12% compared to a real machine. It can simulate arbitrary cache configurations. However, it uses Pin to capture the in-

structions executed by the workload and thus does currently not support full-system emulation [36].

Another way to speed up full-system or cache simulation is by using field-programmable gate arrays (FPGAs), which can be used to simulate many cores simultaneously. The applicability of this is explored in the ProtoFlex [22] system proposed by Van Chu [46]. Approaches using FPGAs are shown to give a high-performance increase, which unfortunately comes at the cost of significantly reduced configurability compared to a software-based approach.

2.2 System design

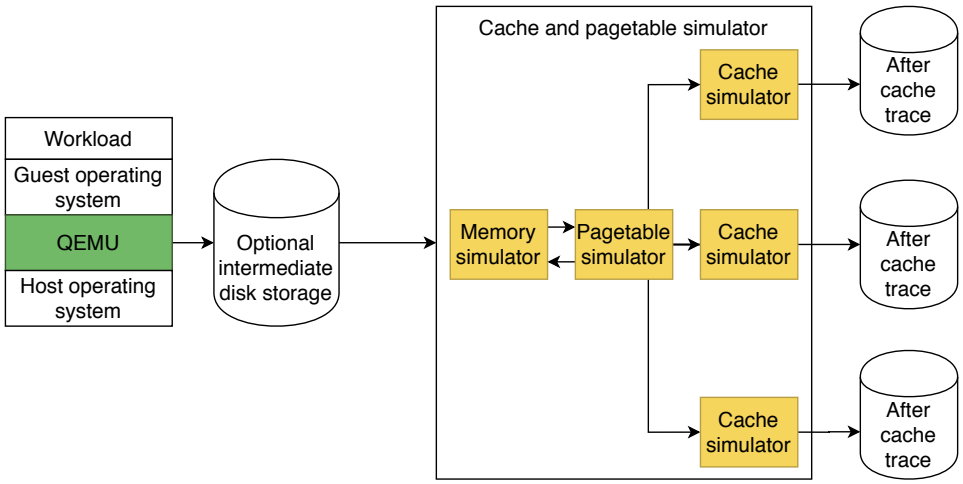


Figure 2.4: High level overview of the proposed framework

Because no simulator is available which fulfils all the requirements for full system after-cache memory tracing, a hybrid approach is proposed. The hybrid approach can combine the performance and flexibility of full-system simulators with the accuracy of simulating caches in a dedicated cache simulator. An overview of the hybrid approach can be seen in Figure 2.4. Cache traces of all accesses before the cache will be gathered using QEMU, shown in green, and the output of QEMU will be used as input for a custom developed cache simulator. Using this approach, the existing implementation of QEMU can be leveraged, which is able to simulate a wide variety of CPU architectures. Furthermore, if the modifications made to QEMU are kept to a minimum, the simulator can use future improvements made by the active QEMU community.

The cache simulator, shown in yellow in Figure 2.4, can be used to convert the traces to after-cache traces after QEMU has finished. Using a simulator independent format for trace files to allow both the modified QEMU and cache simulator to be used stand-alone, for example in a case where QEMU is replaced by gem5 or traces are generated by binary instrumentation tools like Pin. Furthermore, this

modular approach allows evaluation of the system as a whole and both components in isolation to compare them to existing solutions. Another mode of operation is implemented where QEMU and the cache simulator run in parallel and the traces generated by QEMU are directly used as input for the cache simulator without writing the traces to a disk. To obtain minimum slowdown, as described by research question 1, the cache simulator should, in this case, be fast enough to handle the traces generated by QEMU to ensure it does not cause a bottleneck. To answer research question 3 multiple cache simulators with different configurations must be able to run in parallel. To be able to perform correct cache simulation, the physical address for each traced memory access has to be known. The translation from virtual to physical addresses is done by simulating the guest memory and page tables in the page table and cache simulator. To be able to perform this translation in the cache simulator, the data written for memory write accesses is included in the access trace. This data, together with the updates to the control register values of the guest allows the cache simulator to generate traces regarding the page tables.

2.3 Benchmarks

To evaluate system performance and compare it to other solutions benchmark workloads can be used. In this thesis, the Princeton Application Repository for Shared-Memory Computers (PARSEC) [17] is used. This is a widely used benchmark suite which consists of real-world multi-threaded workloads of which several will be used to validate the correctness and measure the performance of the system. The benchmarks are executed on a guest system running Ubuntu 18.04.03LTS and version 5.0.0 of the Linux kernel. The official precompiled binaries available on the PARSEC website are used [10]. Another advantage of using the PARSEC benchmark is that information like the total amount of instructions, memory reads, and memory writes per benchmark is known [18]. Three main workloads with different memory characteristics are chosen from PARSEC suite to perform benchmarks of the performance of the system during implementation. These three are chosen to represent workloads with different memory characteristics regarding read to write ratio and working set size:

1. **Blackscholes** is the smallest benchmark of the PARSEC suite regarding the number of memory accesses. It has lower memory traffic than the other benchmarks and a small working set of 2MiB for the simlarge input size which potentially fits completely in the simulated cache [18].

2. **Streamcluster** has a high read to write ratio, on average more than 4 bytes of memory traffic per instruction, only less than 1% of all memory accesses are writes [18].

3. **Freqmine** has approximately 4 bytes of memory traffic per instruction and one-third of all memory traffic is write traffic [18]. Furthermore, it has significantly more synchronisation primitives between threads than the other

benchmarks. These primitives will result in context switches. Therefore this will be an interesting test for page table simulation.

Apart from the experiments executed on the DAS-5, all experiments are performed on a host system with an Intel 4790k processor, 16GiB memory and a high speed 1TB M.2 PCIe solid-state drive for trace storage. This system runs Manjaro Linux version 19.0.0.

Chapter 3

Trace gathering

The first part of the hybrid approach for tracing after-cache traces is gathering all memory accesses in QEMU. To answer research question 1 an approach for tracing all memory accesses in QEMU is proposed in this chapter This approach aims for a minimum performance decrease of tracing memory accesses. The memory of the guest simulated in QEMU can be accessed in different ways. These are described in detail. To be able to accurately trace all cache accesses QEMU needs to be modified, this modified version of QEMU will be called QMEMU. By default, QEMU supports two implementations to keep track of events happening in the system, plugins and tracing. Both of these approaches will be explored. For cache simulation, the physical address of each access has to be known. Therefore the feasibility of tracing physical addresses in QEMU is reviewed. Furthermore, a way to trace the page tables of a system should also be devised to answer research question 4.

3.1 Memory accesses in QEMU

QEMU uses a tiny code generator (TCG) to translate guest instructions to host instructions. The guest instructions are first translated by the frontend to a universal intermediate language consisting of TCG operations. These TCG operations can then be translated to host operations by one of the backends. An example of this translation is shown in Table 3.1. To speed up simulation QEMU does not translate the code instruction by instruction but divides the guest code into basic blocks. This can also be seen in Table 3.1.a where the last instruction of the block is *ret*. These basic blocks of guest instructions are translated to blocks of host instructions by the backend. QEMU maintains a cache of these translated blocks to prevent translating the same basic block many times in a short period.

When no direct equivalent frontend operation exists for one of the guest instructions, it can be simulated by translating it to multiple frontend operations. A single guest swap instruction can be translated into multiple load and store instructions, for example. Another option is to use a helper function. Helper functions can be

RISC-V instructions	QEMU intermediate ops	x86_64 instructions

	`ld_i32 tmp0,env,$0xfffffffffffffff8`
	`movi_i32 tmp1,$0x0`
	`brcond_i32 tmp0,tmp1,lt,$L0`	

`0x0000000000010458: 4781 mv a5,zero`	`---- 0000000000010458`
	`movi_i64 tmp2,$0x0`
	`mov_i64 x15/a5,tmp2`
		`0x55a64ae58340: movl -8(%rbp),%ebx`
		`0x55a64ae58343: testl %ebx,%ebx`
		`0x55a64ae58345: jl 0x55a64ae58386`
		`0x55a64ae5834b: xorl %ebx,%ebx`
		`0x55a64ae5834d: movq %rbx,0x78(%rbp)`
		`0x55a64ae58351: movq %rbx,0x50(%rbp)`
		`0x55a64ae58355: movq 0x10(%rbp),%rbx`
		`0x55a64ae58359: leaq 0x18(%rbx),%r12`
		`0x55a64ae5835d: movq 0(%r12),%r12`

`0x000000000001045a: 853e mv a0,a5`	`---- 000000000001045a`
	`mov_i64 tmp2,x15/a5`
	`mov_i64 x10/a0,tmp2`
		`0x55a64ae58361: movq %r12,0x40(%rbp)`
		`0x55a64ae58365: addq $0x20,%rbx`
		`0x55a64ae58369: movq %rbx,0x10(%rbp)`

`0x000000000001045c: 6462 ld s0,24(sp)`	`---- 000000000001045c`
	`mov_i64 tmp2,x2/sp`
	`movi_i64 tmp4,$0x18`
	`add_i64 tmp2,tmp2,tmp4`
	`qemu_ld_i64 tmp3,tmp2,leq,0`
	`mov_i64 x8/s0,tmp3`
		`0x55a64ae5836d: movq 8(%rbp),%rbx`
		`0x55a64ae58371: andq $0xfffffffffffffffe,%rbx`
		`0x55a64ae58375: movq %rbx,0x200(%rbp)`
		`0x55a64ae5837c: movq %rbp,%rdi`
		`0x55a64ae5837f: callq 0x55a64acd81c0`

`0x000000000001045e: 6105 addi sp,sp,32`	`---- 000000000001045e`
	`mov_i64 tmp2,x2/sp`
	`movi_i64 tmp3,$0x20`
	`add_i64 tmp2,tmp2,tmp3`
	`mov_i64 x2/sp,tmp2`
		`0x55a64ae58384: jmpq *%rax`
		`0x55a64ae58386: leaq -0x10a(%rip),%rax`
		`0x55a64ae5838d: jmp 0x55a64ae28018`

`0x0000000000010460: 8082 ret`	`---- 0000000000010460`
	`mov_i64 pc,x1/ra`
	`movi_i64 tmp3,$0xfffffffffffffffe`
	`and_i64 pc,pc,tmp3`
	`call lookup_tb_ptr,$0x6,$1,tmp3,env`
	`goto_ptr tmp3`
	`set_label $L0`
	`exit_tb $0x55a64ae58283`	

a.	*b.*	*c.*

Table 3.1: QEMU translation from RISC-V to x86_64

defined in the QEMU source code and can be called using the TCG instruction *call*. In the QEMU frontend operations, there is no instruction equivalent to the RISC-V *ret* instruction defined. In Table 3.1.b it can be seen that the helper function *lookup_tb_ptr* function is called to simulate the RISC-V *ret* instruction.

Because of how memory is simulated internally in QEMU, the simulated guest memory can be accessed in three different ways. Each of these will be described in detail in one of the subsections below.

3.1.1 Load and store instructions

The most straightforward way in which the guest memory is accessed is by simulated guest instructions which access the memory such as load and store TCG instructions. These will be translated to instructions for loading and storing memory: *ld[x]s_i[n]* and *st[x]_i[n]* here *x* indicates the amount of bytes to be read and *n* is the size of the guest registers. The TCG operations for these instructions can be generated by calling the functions *tcg_gen_ld...* or *tcg_gen_st...*. To trace the memory accesses, these functions can be modified to insert additional instructions which call a helper function that registers the memory access.

3.1.2 Memory accesses in helper functions

The second way in which guest memory can be accessed is from helper functions which are called from the generated code in the translated blocks. To access guest memory from these helper functions QEMU has multiple memory accessor functions of which *cpu_ld[x]_data* and *cpu_st[x]_data* are examples. By simply instrumenting all accessor functions, all memory accesses performed by helper functions can be traced.

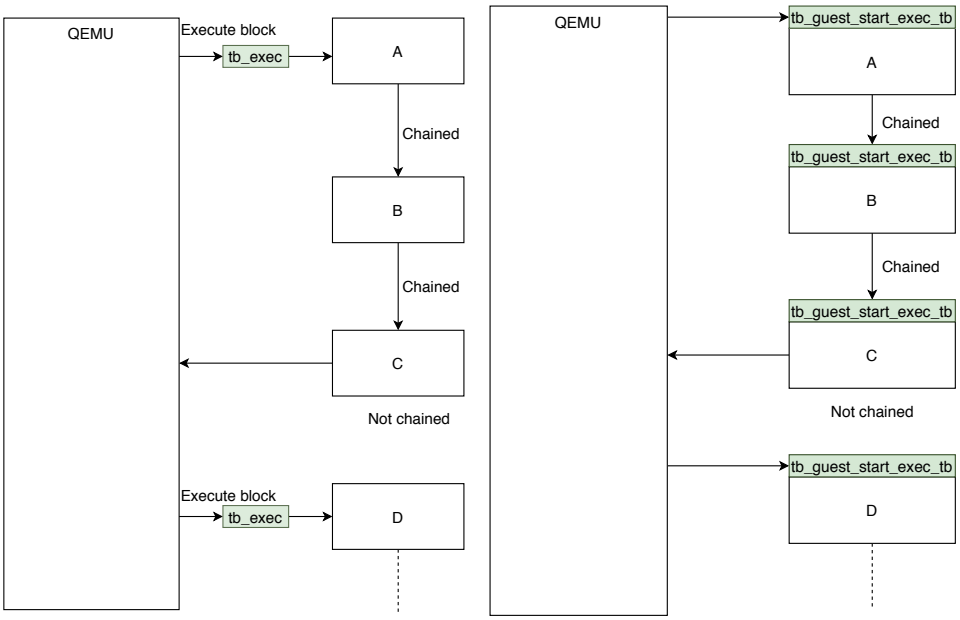
3.1.3 Instruction fetches

The final way in which guest memory can be accessed in QEMU is by instruction fetching. In an actual system, this done by the CPU itself based on the value of the program counter. The program counter indicates the memory location of the instruction to be executed by the processor and will be increased after every execution. Jump instructions explicitly modify the program counter to jump to different parts of a program. Modern CPUs will prefetch the upcoming instructions to ensure the processor does not have to wait for the next instruction to be loaded from memory after the previous one was executed.

In QEMU, when executing translated blocks, keeping track of the guest program counter and fetching instructions is done by the host. QEMU caches the translated blocks. On a cache miss, QEMU will load the required instructions from the guest memory and translate them to the corresponding host instructions. To trace instruction fetches the function used when a cache miss occurs could be instrumented. Because of the size of the translated block cache, most of the translated blocks can be fetched from this cache instead of the guest memory. Therefore potentially many instruction fetches will not be traced. Another optimisation which QEMU employs is block chaining. Chaining blocks is done by extending each block with instructions to jump to the following block. Therefore the host executing the instructions in the translated block will jump to the start of the next block by itself. As a result, the translated block cache does not have to be explicitly searched by QEMU for the next block after the execution of a block. This option can be disabled by using the QEMU *'nochain'* option, but this slows down simulation significantly by almost four times.

Instruction fetches can be traced by instrumenting each guest instruction to trace the value of the program counter when the instruction is executed. Instrumenting each instruction will cause a significant overhead because a call to a helper function has to be performed for each instruction.

Because the guest code is divided into blocks delimited by jump instructions, it is guaranteed that all instructions of a block will be executed and thus fetched. Therefore it suffices to trace the start of each translated block. By default, QEMU has a trace event *exec_tb* that does exactly this. However, this event is only traced when explicitly starting execution of a translated block and will not be traced when a block is executed because it was chained to the end of another block. Thus not



(a) Translated block tracing using the default *exec_tb* event

(b) Translated block tracing using the new *guest_start_exec_tb* event

Figure 3.1: Difference between the *exec_tb* and *guest_start_exec_tb* trace events, traced events are shown in green

all translated blocks will be traced; this is illustrated in Figure 3.1a. When blocks are chained only the start of the first block will be traced unless the *nochain* option is enabled.

To be able to trace all instruction fetches while minimising the performance decrease caused by this QEMU is modified. To ensure that the start of execution of each chained block is traced a call to a helper function is inserted at the start of each translated block. This is illustrated in Figure 3.1, in this figure the difference between the default *exec_tb* trace event and the newly implemented *tb_guest_start_exec_tb* trace event can be seen. With the new implementation regardless if blocks are chained, the helper function which traces the instruction fetches will be called using the *tb_guest_start_exec_tb* trace event. In this helper function, the memory address of the first instruction and the size of the block can be traced. This helper function in turn will call a trace event called *guest_start_exec_tb*.

To be able to determine which memory addresses were accessed for each executed translated block, the program counter and size of the guest instructions of the block have to be traced. For each translated block the size of the basic block could be stored along with the translated block in the cache. Unfortunately, of how block translation is implemented in QEMU, the size of the basic block not known when the prefix instructions for the block are generated. Inserting the call to the helper function at the start of the block after the rest of the instructions of

the block have been translated would require a modification of the code for all supported QEMU guest platforms which is not desirable. If the call to the helper function is inserted at the end of each block, it will not get called when the final jump instruction jumps to another block. However, as opposed to when the block is being translated, when the inserted helper call is performed, the complete block has been translated. Thus at this moment, the size is available in the cache entry for the block. Therefore by adding a pointer referencing the translated block as a parameter to the helper function, the size of the block can be found when the helper function is called instead of providing it as a hard-coded parameter to the helper function when a block is translated.

By comparing the debug output of QEMU to the output obtained using the helper function call, it can be validated that this method of instruction fetch tracing works correctly. The required debug output can be enabled running QEMU with the *exec* and *nochain* options. The *exec* prints debug information before each translated block and *nochain* prevents chaining translated blocks. Together these options make QEMU print every guest code block before it is executed. A snippet containing the implemented trace output using the log tracing backend and the debug output can be seen in Table 3.2. For the first block that is executed a program counter value of 400000114 is logged together with a size of 63 bytes. The same value is found by counting the number of instruction bytes in the first block of the debug output results in the same 63 bytes. It can also be seen that the logged value of the program counter is correct.

Accesses of instructions performed by the CPU are always memory reads and have a dedicated L1 cache. Therefore if one is purely interested in tracing write accesses or accesses which happen because of guest load and store instructions, these instruction fetches do not have to be changed. Leveraging the configurability of QEMU, the additional trace event can be enabled and disabled when compiling QEMU. Like all other trace events, when disabled at compile-time, there will be no impact on system performance.

3.2 QEMU plugins

Besides the tracing functionality, QEMU also provides a plugin functionality. When plugins are enabled, dynamic libraries can be provided to QEMU at runtime. These libraries can register callback functions using the provided QEMU plugins API. Multiple callback functions can be registered for certain events that occur in the QEMU system. One of the events to which a plugin can register its callback functions is to the memory access event. The provided callback function will then be called each time QEMU executes a load or store backend TCG instruction. The implementation of plugins is quite similar to how trace events are implemented in QEMU. For each translated memory instruction, an additional TCG helper function call is inserted. The plugin will call all registered callbacks for the event Because of the overhead of QEMU plugins, the performance of tracing all memory accesses

```
....
Instruction fetch for pc:4000001149 of size: 63 for cpu: 0
Instruction fetch for pc:4000001188 of size:  2 for cpu: 0
Instruction fetch for pc:40000011ab of size: 12 for cpu: 0
Instruction fetch for pc:400000118a of size: 45 for cpu: 0
...
```

a.

```
...
----
0x4000001149:  55                      pushq    %rbp
0x400000114a:  48 89 e5                movq     %rsp, %rbp
0x400000114d:  48 81 ec 20 08 00 00    subq     $0x820, %rsp
0x4000001154:  64 48 8b 04 25 28 00 00 movq     %fs:0x28, %rax
0x400000115c:  00
0x400000115d:  48 89 45 f8             movq     %rax, -8(%rbp)
0x4000001161:  31 c0                   xorl     %eax, %eax
0x4000001163:  c7 85 ec f7 ff ff 00 00 movl     $0, -0x814(%rbp)
0x400000116b:  00 00
0x400000116d:  48 8d 85 f0 f7 ff ff    leaq     -0x810(%rbp), %rax
0x4000001174:  48 89 c6                movq     %rax, %rsi
0x4000001177:  48 8d 3d 86 0e 00 00    leaq     0xe86(%rip), %rdi
0x400000117e:  b8 00 00 00 00          movl     $0, %eax
0x4000001183:  e8 b8 fe ff ff          callq    0x4000001040
----
0x4000001188:  eb 21                   jmp      0x40000011ab
----
0x40000011ab:  81 bd ec f7 ff ff 01    cmpl     $0x1ff, -0x814(%rbp)
0x40000011b3:  00 00
0x40000011b5:  76 d3                   jbe      0x400000118a
----
0x400000118a:  83 85 ec f7 ff ff 01    addl     $1, -0x814(%rbp)
0x4000001191:  b8 00 02 00 00          movl     $0x200, %eax
0x4000001196:  2b 85 ec f7 ff ff       subl     -0x814(%rbp), %eax
0x400000119c:  89 c2                   movl     %eax, %edx
0x400000119e:  8b 85 ec f7 ff ff       movl     -0x814(%rbp), %eax
0x40000011a4:  89 94 85 f0 f7 ff ff    movl
%edx, -0x810(%rbp, %rax, 4)
0x40000011ab:  81 bd ec f7 ff ff 01    cmpl     $0x1ff, -0x814(%rbp)
0x40000011b3:  00 00
0x40000011b5:  76 d3                   jbe      0x400000118a
----
....
```

b.

Figure 3.2: Instruction trace information obtained (a) by the instruction tracing implementation and (b) using QEMU debugging

Name	Output	Advantages	Disadvantages
Log	*stderr*	Human readable output	Large overhead Usage of slow stderr
Simpletrace	File	Output format easily modifiable High throughput	Discards events when buffer is full
Ftrace	Ftrace		Large overhead, unsuitable for high frequency events
Syslog	syslog	Able to compress events	Ratelimits high frequency events by discarding them
UST	LTTng	Supports a blocking mode	Unable to trace TCG trace events Plain text or LTTng specific output format

Figure 3.3: Comparison of different QEMU trace backends

is lower compared to using QEMU tracing. It takes a simple implementation of memory tracing using plugins where each time a memory access occurs the only virtual address is written to a file 1221 seconds to execute the blackscholes simlarge benchmark for two simulated CPU cores. This is more than the 1049 seconds it takes to trace all required information instead of just the virtual address using the same benchmark with the not optimised version of the QEMU simple tracing backend. This is a difference of 20%. Note that here the implementation using tracing records more information than the plugin implementation which records only the virtual address for each access. Furthermore, this is a not optimised version of tracing. This increased runtime together with the advantage of being able to use the already existing infrastructure for tracing, which allows also tracing different, tracing events using QEMU tracing is preferred over using QEMU plugins.

3.3 QEMU Trace backends

As described in the previous chapter, QEMU can be used in two different modes. The one most suitable to be used for memory tracing is TCG mode. Apart from tracing events that happen in the QEMU host code, which can be added by inserting function calls in the host source code using the TCG tracing functionality translated guest code can be instrumented with additional instructions. In QEMU tracing implementation, different trace events can be enabled or disabled at runtime. When an enabled trace event occurs in the system, it will be forwarded to a trace backend which handles the events. Out of the box, QEMU supports various backends, a comparison of these is shown in 3.3.

Apart from these backends, custom backends can be implemented if the user wishes. The log backend is mainly meant to be used for debugging purposes and apart from the fact that the output is human-readable, has no advantage over the simple trace backend. The simple trace backend writes the output as binary data and has a significantly higher performance than the log backend which writes to

the relatively slow stderr descriptor. The output of Ftrace is more verbose than the log or simple trace backends; this causes significant overhead, thus for events that happen very frequently in the system, like memory accesses, this backend is unsuitable. Syslog can compress and rate limit certain events and is therefore unsuitable when all events have to be captured. Similar to Syslog, LTTng can compress but also discards trace events since by default it prefers performance over completeness. However, LTTng can be configured to run in blocking mode in which no events will be lost. Unfortunately, it was discovered that TCG type trace events could not be traced when using LLTng. These are trace events which can be inserted in generated guest code. The *guest_mem_before* event is one of those events. Therefore for memory access tracing the simple trace backend most suitable however it requires a minor modification that allows it to run in a blocking mode,

3.4 QEMU Simple trace backend

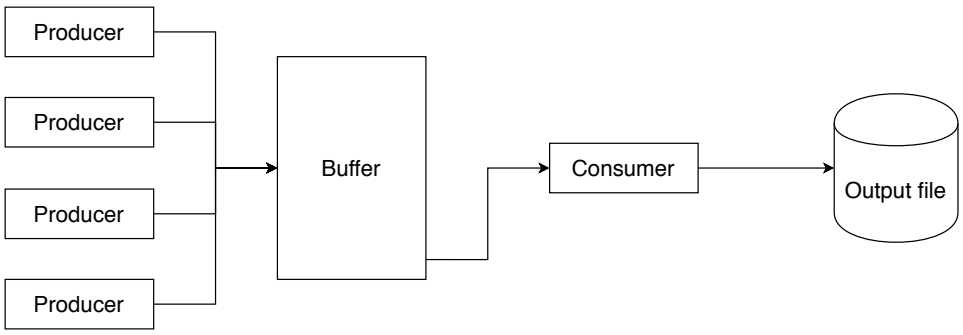


Figure 3.4: The simple trace backend producer consumer model

To make the QEMU trace backend fully suitable for memory tracing several modifications are made. The simple trace backend implements a producer-consumer model, a diagram of this can be seen in Figure 3.4. Different producer threads can put trace events in a trace buffer. A single consumer thread reads the trace events from this buffer and writes them to a file. If the consumer cannot keep up with the producer threads, the buffer fills up. When the buffer is completely full, the producers will discard any trace events they receive. One of the main requirements for the captured traces is that they should be complete. Traces are complete when every memory access is captured. Furthermore, to reduce the slowdown caused by tracing memory accesses the size of the trace events is reduced.

3.4.1 Dropped trace events

By default, the QEMU trace backends are not designed to handle a high amount of trace events. The simple trace backend uses a buffer for the trace events to

which different threads can write events. A single consumer thread reads events from the buffer one by one and outputs them to the specified file. The simple trace backend is modified to prevent discarding trace events. By this modification, the producer threads block when the buffer is full until space in the buffer becomes available. This blocking implementation will ensure the gathered traces contain all memory accesses made by the guest system. When the consumer thread can not keep up with the threads filling the buffer, it will eventually fill completely. Then completely full events will be dropped by the producers. To allow page table simulation and provide accurate cache simulation, it is required that complete cache traces of every guest memory access are gathered.

		Runtime (seconds)	
Mode	**Output location**	**Average**	**Standard dev.**
Non-blocking	*/dev/null*	475	6.2
Non-blocking	file	255	21.8
Blocking	*/dev/null*	515	5.0
Blocking	file	1049	25.2

Table 3.2: The average runtime of 5 executions of the PARSEC blackscholes simlarge benchmark with the *guest_mem_before_exec* trace event enabled

The simple trace backend has been modified such that events are not dropped when the buffer is full. Instead, the producer threads waits until space in the buffer becomes available instead of simply discarding the event. Table 3.2 shows the performance overhead of the blocking implementation for the PARSEC blackscholes benchmark. These are the results for a QEMU configuration with 8GiB of simulated memory and two simulated CPU cores. The average runtime of 5 executions of the benchmark is taken. The simlarge input was used with the *guest_mem_before_exec* trace event was enabled. When tracing to a file, the benchmark takes 4 times longer to complete when the simple trace backend blocks compared the mode where the tracing backend drops traces. The runtime stays approximately the same when the output is */dev/null*. This can be explained by the fact that when using */dev/null* as output, the consumer thread is fast enough and can keep up with the producer threads filling the buffer. It might seem counter-intuitive that for the non-blocking implementation, output to a file is faster than when using */dev/null* as the output location. However, when using the non-blocking implementation with */dev/null* as output, which is faster than using a file, the trace consumer can write more trace events to the output. This means that the producer threads drop fewer traces, and thus have to acquire locks to the trace buffer more often. Thus instead of the consumer, the producers will be the bottleneck. This causes a faster output location to become slower. Thus a minimal amount of trace events will be dropped, and the speed of the consumer mainly limits the runtime. Conversely, when a trace event is dropped before writing it to the buffering thread can continue instead of waiting for

space in the buffer to become available. Therefore when the trace output location is a file the bottleneck is the consumer thread, it cannot keep up with the producer threads filling the buffer and thus trace events will be dropped by the other threads.

Benchmark	**Blackscholes**	**Streamcluster**	**Freqmine**
	Runtime (seconds)		
No tracing	4.6	5.8	9.3
Tracing to */dev/null*	301.3	624.7	1199.0
Tracing to file	1032.8	1718,2	3114.8
	Accesses per second (millions)		
No tracing	339.8	480.7	575.6
Tracing to */dev/null*	5.21	4.5	4.46
Tracing to file	1.5	1.6	1.7
	Slowdown		
No tracing	1	1	1
Tracing to */dev/null*	65	107	128
Tracing to file	225	296	335

Table 3.3: The performance impact of enabling the *guest_mem_before_exec* trace event in QMEMU with the blocking simple trace implementation for the simmedium benchmark size (average runtimes of 3 executions is shown)

The impact of having a blocking trace buffer for different benchmarks can be found in Table 3.3. The slowdown when tracing to */dev/null* compared to no tracing ranges from 65 to 128 times, this difference is caused by the differences in accesses per second that QEMU can simulate when not tracing. It can be seen that the number of accesses per second that QEMU can handle for different benchmarks stabilises when tracing is enabled. The runtime can be seen to increase roughly two orders of magnitude when using */dev/null* as output compared to no tracing. The runtime increases approximately three times when tracing to a file instead of to outputting the trace to the */dev/null* pipe.

One of the main bottlenecks could be the size of the buffer for the consumer thread. A solution for this would be to increase the size of the buffer. This way, the consumer thread has more time to handle bursts of memory accesses; this will potentially decrease the amount of time the producing threads have to wait for space in the buffer to become available. By default, the buffer is 256KiB the influence of the buffer size on the runtime of different benchmarks with the *simmedium* input can be seen in Table 3.4. For the tested benchmarks, no significant difference in runtime can be seen for larger buffer sizes. Therefore the default buffer size of 256KiB will be used for the experiments in the remainder of this thesis. However, the number of simulated cores and memory access frequency of the benchmark

Buffer size	Runtime (seconds)		
	Blackscholes (simedium)	Streamcluster (simsmall)	Freqmine (simdev)
4KiB	297.4	213.3	191.5
16KiB	297.3	205.6	198.2
64KiB	304.1	207.1	195.7
256KiB	301.1	206.3	184.8
1MiB	305.4	208.2	182.8
4MiB	297.8	200.7	187.0
16MiB	290.7	207.2	183.3
64MiB	294.7	208.9	181.1

Table 3.4: Runtime for different trace buffer sizes for the blackscholes simmedium benchmark in QMEMU with the *guest_mem_before_exec* trace event enabled when using */dev/null* as output for a simulated guest system with 4 cores

can potentially influence the needed buffer size; therefore this buffer size should be configured to suit the workload and amount of simulated cores.

3.4.2 Shared memory

The simple trace backend was modified to allow the usage of POSIX shared memory for the output to achieve higher throughput. This shared memory can be read by another process to be processed. The shared memory region is protected by two semaphores to avoid concurrency problems. When QEMU has filled the complete buffer, it lowers the *write* semaphore and raises the *read* semaphore. The other process reading the trace events is then allowed to read the buffer, when it has finished reading it lowers the *read* semaphore and raises the *write* semaphore. To ensure QEMU is not unnecessarily waiting for the buffer to be filled, the buffer is split into two parts, each protected by a separate pair of semaphores.

3.4.3 Trace size reduction

Because of the large number of memory accesses, the size of each event must be as small as possible. A breakdown of the different parts of a memory trace event can be found in Table 3.5. The trace size can be reduced by modifying the following fields in the binary format:

- **Record type**: The trace contains two types of entries, actual trace events and a mapping from IDs of these trace events to their names. The record type field specifies if the next bytes represent a trace event mapping or an actual tracing type. If the simple trace backend is modified to write the mapping and actual events to two different files, the actual trace only contains a single type of records, and therefore this field can be omitted.

Item	Size (bytes)	Reduced size (bytes)
Record type	8	0
Event ID	8	1
Timestamp	8	2 or 8 [1]
Length	4	0
PID	4	0
CPU	8	1
Address	8	8
Info	8	1
Data	8	0 or 8 [2]
Total	**64**	**14-28**

[1] depending on the difference with the previous timestamp
[2] 0 bytes for read accesses 8 bytes for write accesses

Table 3.5: Output size of a memory write trace event

- **Event ID**: This field is used to indicate which type of trace event follows it. Because only a couple types of trace events are used for memory tracing this field can be reduced to a single byte, a single byte will still allow 256 distinct memory trace events to be enabled at the same time.

- **Length**: The length field is used to specify the total length of the current trace event. Since the length is constant and known for all types of trace events used, this field can be omitted.

- **Timestamp**: The timestamp is specified as a 64-bit integer relative to the start of the simulator. However, it can be changed to the difference between the timestamp of the previous event that was traced and the timestamp of the new event. A more detailed description of this optimisation is given in Subsection 3.4.3.

- **PID**: The process ID (PID) field is set to host the process ID of QEMU. This field is the same for all trace events and can thus be omitted.

- **CPU**: The CPU field indicates in which of the cores of the simulated system the event occurred. In QEMU each simulated core is stored in a memory structure. The host memory address of this structure is stored for each trace item. This address is stored using 8 bytes. The structure contains a CPU index which has the number of the core counting from 0 to n. This index is used instead of the value. When stored using a single byte a maximum of 256 cores can be simulated by QEMU.

- **Info**: The info field contains various pieces of information about the memory access for example whether the access is a read or a write, in QEMU these values are already packed into a 2-byte struct using bit fields but written to the trace as a 64-bit integer. One of these two bytes contains the QEMU MMU index. This value is not needed for cache simulation. Therefore this field can be stored using a single byte.

Timestamp size compression

Size (bits)	Percentage of timestamp differences which fits in the amount of bits
8	62.24%
16	99.99%
32	99.99%
64	100.00%

Table 3.6: Size of the difference between timestamps in bytes for the blackscholes simlarge benchmark

The timestamp field in the trace events specifies the time in nanoseconds at which the trace event occurred since the start of the simulator. This field is stored in the trace output using 8 bytes. However, the difference between the timestamp and the previous timestamp is often relatively small. Therefore a similar approach to the one described by Bao et al. [14] is implemented. The number of bytes required to store this difference was determined using a memory access trace of the blackscholes benchmark with the simlarge input size. In Table 3.6 the difference in timestamps can be found. This table shows many of the time differences can be represented using various amounts of bits. Precisely, this is the percentage of time differences which is less than 2^n where n is the number of bits. From this data, it can be seen that using two bytes to store the timestamp difference is most suitable. Unfortunately, the larger differences between timestamps cannot be truncated or omitted because these have a larger influence on the overall accuracy of the simulator output. Therefore a solution where the most significant bit (MSB) of the *event_id* field of a trace event is used to indicate the size of the timestamp difference. When the MSB of this field is set the timestamp field has a size of 8 bytes when it is not set the delta timestamp field is stored in 2 bytes. When multiple CPUs are simulated, a trace event with an earlier timestamp may arrive at the trace consumer thread later than an event that happened in another CPU with a later timestamp. The second MSB of the *event_id* field is used to indicate whether the timestamp difference is positive or negative.

Assuming 99.99% of the timestamp differences can be stored in a 2-byte value, this approach will reduce the trace size of each item significantly by approximately

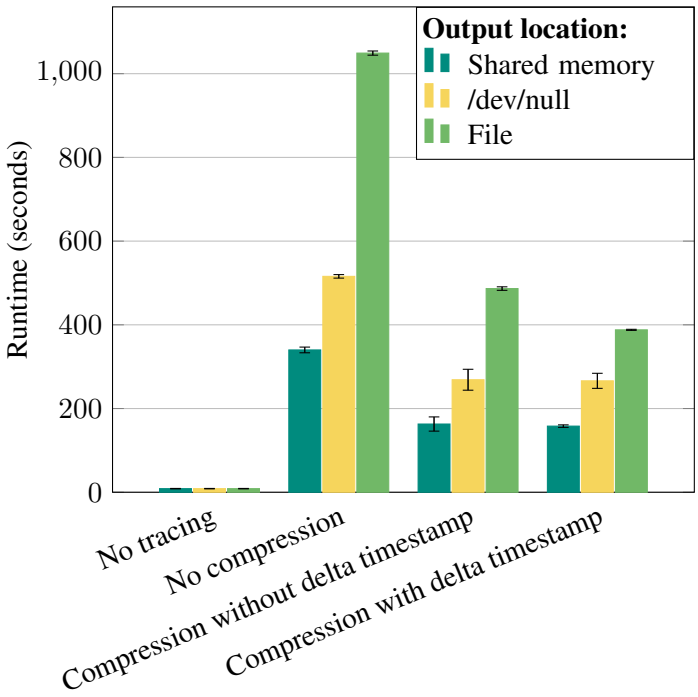


Figure 3.5: The runtime of 5 executions of the QMEMU tracing for the blackscholes simlarge benchmark with the *guest_mem_before* trace event enabled, the black bars indicate the standard deviation values

21%. This compression adds some additional logic to the trace write out functionality in QEMU to determine if a 2 or 8-byte time delta should be written. From Table 3.6 it can also be seen that approximately 62% of the timestamp differences will fit in a single byte value. A distinction between 1 and 2-byte delta timestamps will only reduce the trace size by another 2.8% while adding even more complexity to the logic of writing and reading the trace events. Therefore only a distinction between 2 and 8-byte values is made.

In Figure 3.5, the runtime for the blackscholes benchmark with the simlarge input can be seen with various optimisations enabled. The average runtime of 5 executions in seconds together with the minimum and maximum runtime is expressed on the y-axis. Various optimisations are shown on the x-axis beginning with the runtime with tracing disabled, the runtime without any of the described optimisations, the runtime with the reduced trace size but without the delta timestamp size compression and finally the runtime with the timestamp size compression enabled. From this graph, it can be seen that with the modifications to the simple trace backend, there is a reduction in the runtime of approximately 2.5 times when tracing to a file. Because the optimisations reduce the trace size, the speedup is lower when */dev/null* or shared memory is used for the output since these are not limited by the bandwidth of the storage device. However, a comparison of the av-

erage runtime of the benchmark when using a file to store the trace output shows a decrease from 485 to 388 seconds this is a reduction of 20%. When */dev/null* or shared memory was used as the output location, the average runtime does not change significantly when using delta timestamps. Indicating that the benefit of the smaller trace file still outweighs the slightly increased complexity in the consumer thread when the bandwidth of the output medium is not the bottleneck.

3.5 Physical address tracing

To be able to simulate CPU caches correctly together with the virtual address, the physical address of each memory access has to be traced. In a physical CPU, the address translation is done by the memory management unit (MMU). QEMU simulates the guest MMU and maintains a cache of translated virtual to physical addresses. This cache is called the translation lookaside buffer (TLB). When a memory location is accessed, the virtual address is looked up in the TLB, if it contains the physical address it can be used directly. When the corresponding physical address is not found in the TLB the page table has to be searched by the QEMU MMU, the result will then be placed in the TLB. The trace events currently used only record the guest virtual address for memory access. When using TCG the physical address is not known at the location the code where memory accesses are traced. However, for a virtual address, the TLB can be used to look up the corresponding physical address. The trace events used to trace memory accesses are usually called before the access is simulated, to ensure the TLB contains an entry for the virtual address tracing is moved to after performing the memory access. When performing a TLB lookup directly after the access, will always contain the virtual address. Because it will be filled by a page table walk if the TLB did not contain an entry before the access was simulated.

To do this lookup a function *lookup_tlb_informational* was implemented which looks up a physical address in the TLB but will not do a page table walk upon a miss. For the memory accesses performed by helper functions, this function can simply be called and the output can be used for the trace event. For the tracing function calls inserted in the translated blocks at translation time, an additional call to this function has to be inserted in the translated block. An example of this can be seen in Table 3.7. During translation, a call to the helper function will be inserted in the translated instructions. This helper function takes several QEMU virtual registers as inputs and outputs the physical address in another virtual register. The virtual register is used as input to the tracing function helper call. Therefore, the translation will add additional overhead in the slowest path of tracing. These changes increase the average runtime for tracing the blackscholes simlarge benchmark from 228 to 391 seconds, this is an increase in runtime of 1.7 times. Because of this slowdown, this method of gathering physical addresses will not be used.

Without physical address tracing	With physical address tracing

`trace_guest_mem_store_before_tcg(`
`  tcg_ctx->cpu, cpu_env, addr, info, val);`
	`TCGv physaddr = tcg_temp_new();`
	`TCGv_i32 mmu_idx_i32 = tcg_const_i32(info >> 8);`
	`TCGv_i32 write_i32 = tcg_const_i32(1);`
	`gen_helper_lookup_tlb_informational(physaddr,`
	`  cpu_env, mmu_idx_i32, write_i32, addr);`
	`trace_guest_mem_store_before_tcg(tcg_ctx->cpu,`
	`  cpu_env, addr, physaddr, info, val);`
	`tcg_temp_free(physaddr);`
	`tcg_temp_free_i32(mmu_idx_i32);`
	`tcg_temp_free_i32(write_i32);`

a.	*b.*

Table 3.7: Insertion of physical address tracing helper function

3.6 Page table tracing

To be able to acquire the physical addresses memory accesses, the page tables of the guest will be simulated outside QMEMU. Furthermore, as described in the previous chapter, the page table can provide useful information for analysing memory traces. Therefore the framework should be able to trace contents of the page mapping per process as stored in the page tables. In the cache simulator, the virtual addresses will be translated to physical addresses, to do this the MMU of the guest system has to be simulated. Since the implementation of the page tables is guest architecture-dependent, an implementation will be proposed for the x86 platform. Regardless of this dependency, the general idea of tracing page table modifications is similar for various other architectures such as ARM.

3.6.1 Page table pointer tracing

The x86 platform has multiple control registers which determine the paging behaviour of the system. If the *"paging enabled"* bit of the first control register (CR0) is 0, the MMU will use physical addressing. The operating system uses this mode during boot for setting up the initial page table, for example. Conversely if the *"paging enabled"* bit is 1 the CPU uses the page table to translate addresses. The MMU will use the value of the fourth control register (CR3) to determine the memory address of the first level of the page table it should use. The fifth control register (CR4) contains additional information about the paging scheme used like the size of physical addresses and whether the page table has 4 or 5 levels.

On a context switch, the value of CR3 is updated to point to the page table of the newly assigned process. This value of CR3 could be stored in the access stored for each memory access. The value of CR3 is updated infrequently compared to

the number of memory addresses. Therefore, it will be more efficient to trace only modifications of this register by the guest. A minor downside of this is that the current value of CR3 has to maintained in the cache simulator. To be able to simulate paging correctly modifications to the values of CR0 and CR4 also have to be traced.

Tracing modifications to these registers can be achieved by adding a new trace event to QEMU, the *guest_update_cr* event which contains the id of the CPU, the index of the updated control register and the updated value. As described previously QEMU translates guest instructions in sequences of backend operations which can be executed on the host. It keeps a cache of these translated blocks for reuse in the future. Some complex operations can not easily be translated to backend instructions. In these cases, a helper function can be called to simulate these instructions. For the x86 target updating the CR values is one of these operations for which the backend inserts a call to a helper function. This makes tracing this event trivial, only this helper function has to be modified to call the trace function. Because the *guest_update_cr* event is called very infrequently, there is no noticeable increase in runtime caused by enabling this trace event.

3.6.2 Data tracing

As described previously, when a memory write is traced, the data of write request has to be included in the trace event. The data that is read for a read access performed by the guest is not needed. By default the data is not included in the QEMU *guest_mem_before* event, therefore this event is split in two new trace events, *guest_mem_read_before* and *guest_mem_write_before*. Apart from the additional written data which is stored for *guest_mem_write_before* these trace events are similar to the *guest_mem_before* event. These new trace events are inserted in all locations where the *guest_mem_before* event is traced. Together these two new events provide all trace information which is also provided by the *guest_mem_before*. The *guest_mem_before* event can be disabled at compile-time to reduce its performance penalty. Apart from the slight increase in trace event size, tracing performance is not significantly impacted by this change. These new events cause the average runtime for the blackscholes simlarge benchmark to increase from 158 to 161 seconds when using shared memory for the output.

3.7 Discussion

In this chapter, different ways to trace all memory accesses and the state of the page tables of a system are shown while keeping the number of changes required to QEMU to a minimum. To answer research question 1 it is shown that memory tracing using the simple trace backend modified to output traces to a shared memory buffer is the most efficient method to trace all accesses. In total, an average slowdown of 44 times is achieved when tracing all accesses. Experiments show a

runtime increase of 70% when tracing physical addresses compared to tracing virtual addresses. Therefore to answer research question 1 physical access translation should happen outside the critical path for simulation. Thus virtual to physical address translation should be done in the other part of the hybrid framework, the cache simulator. To be able to do this translation, the state of the page tables has to be known in the cache simulator. The state of the guest page tables is traced by keeping track of modifications to the control register values as well as the data for all memory write operations. This approach also answers research question 4 about how page tables can be traced to reduce the semantic gap. The page tables can be traced by reconstructing the guest memory in which these tables reside in the cache simulator.

Chapter 4

Cache and page table simulation

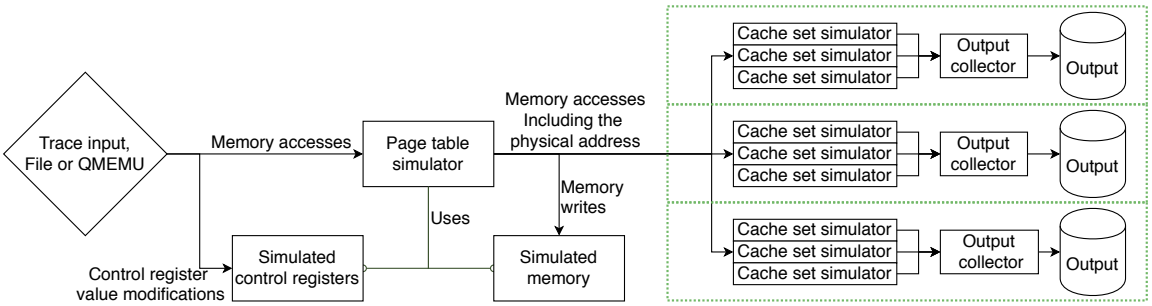


Figure 4.1: High level overview of the cache and page table simulator

The second part of the framework consists of a cache and page table simulator. The primary purpose of this simulator is to provide a solution to the problem posed by research question 2 about how caches can efficiently be simulated. Various systems to simulate CPU caches are available. These approaches are dependent on a specific trace source, like CMP$im [33], do not support simulating multi-core CPUs with coherency protocols, like Dinero IV [1], or are hard to configure like FPGA based approaches like the one proposed by Schneider et al. [45]. Cycle-accurate full system simulators like gem5 can also be used, but their complexity significantly increases simulation time [20]. Their complexity is needed to provide a higher output accuracy with regards to, for example, timing and energy usage. However, when simulating large real-world benchmarks because of the scale of the traces, which can contain tens of billions of accesses, there is little advantage to this increased accuracy. No existing simulator was found which meets all the requirements.

To be able to simulate the caches correctly, the physical address of each cache access has to be known. As described in Section 3.5 QEMU slows down significantly when tracing the physical addresses instead of the virtual addresses. Therefore this translation should be done outside the critical path. Tracing the guest page tables allows this translation to be done in the cache and page table simulator. This

approach also helps answer research question 4 about closing the semantic gap. Furthermore, this allows simulating caches with mixed indexing which no publicly available simulator can do [20].

Because no such system is available, a new system is designed which can do this translation. This system should have minimal impact on the performance of the full-system simulator as required by research question 1. To answer research question 3 the designed simulator should be able to simulate multiple caches in parallel. The designed system parallelises cache simulation for a single cache hierarchy and allows the simulation of multiple hierarchies to be distributed among different machines to increase performance. An overview of the system can be seen in Figure 4.1.

4.1 Background

4.1.1 Caches

Computer caches generally consist of multiple levels of increasing size and decreasing speed. Various policies and techniques are used in caches on different systems.

Replacement policy

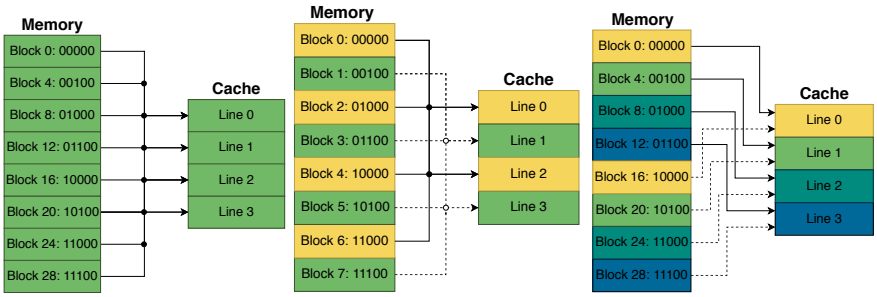
Because the cache has a limited size, there needs to be a policy to determine which items to keep and which to evict when the cache is full. Usually, the least recently used (LRU), least frequently used (LFU), or first in first out (FIFO) policies or a hybrid of these is used.

Write policies

Compared to read-only caches writable caches are harder to implement. The most straightforward way is to have a cache where on each memory write, the value in the main memory is directly updated as well. This type of cache is called a write-through cache and generally has a high latency for writes. Another type of cache is the write-back cache where the state of the cache is not directly mirrored in the main memory. A write-back cache marks entries in the cache as dirty when they are modified. Upon eviction of a dirty cache line, the cache writes the modified value back to memory. Hence the name write-back cache. This increases the implementation complexity of the cache but reduces the write latency compared to a write-through cache.

Placement policies

To reduce the hardware implementation complexity, and thus the physical amount of space on the CPU, caches use a so-called placement policy. This policy limits the number of places in the cache where a memory block can be stored.



(a) Fully associative
3 bit tag
0 bit index
2 bit offset

(b) 2-way set associative
2 bit tag
1 bit index
2 bit offset

(c) Direct mapped
1 bit tag
2 bit index
2 bit offset

Figure 4.2: Different cache placement policies

These policies are implemented by splitting the memory address in an index, tag and offset. Cache lines typically have a size of 8 to 64 bytes. The offset identifies the byte offset within the cache line. The tag is stored along with the contents of the cache line in the cache. The tag is used to identify which memory block is currently present in the cache line. The index is used to identify the cache lines in which a memory block could be stored. The most intuitive way to implement a cache is to have a policy which maps each memory address to exactly one cache line, a so-called direct-mapped cache. An example of a direct-mapped cache can be seen in Figure 4.2c. This approach is easy to implement, fast and energy-efficient. Unfortunately, it is also the worst-performing placement policy. Since there is only one location in which each memory line can be stored, the choice is to either evict the already resident block with the same index or not to store the memory block that was just accessed.

The opposite of a direct-mapped cache is a to make it possible for each memory block to be stored in any location of the cache, called a fully-associative cache. In Figure 4.2a an example of a fully associative cache can be seen. This policy has the highest cache hit ratio and provides better utilisation of the available cache memory. The disadvantage is that when the CPU wants to do a lookup of an entry in the cache, all cache lines have to be searched since the memory block can be stored in any of the cache lines. This significantly increases the implementation complexity of such cache.

Most modern systems take a hybrid approach between the direct-mapped cache and a fully-associative cache, a set-associative cache. Examples of this are the recent AMD [24], ARM [12], and Intel [23] CPUS. In this type of cache, each memory block can be stored one of n different locations. An illustration of a set-associative cache is provided in Figure 4.2b. A set-associative cache consists of m sets which each contain n cache lines. The index of the memory block determines in which set the memory block is stored. When looking up a memory block in

the cache, only the lines contained in the set which corresponds to the index of the block have to be searched. Eviction policies are applied within a set when a cache miss occurs, which increases the hit ratio. A 1-way set-associative cache is the same as a direct-mapped cache. Contrarily, for a cache with k lines a k-way set-associative cache is a fully-associative cache.

Inclusion policy

For multi-level caches, multiple different inclusion policies can be used. An inclusion policy specifies whether blocks contained in a level of the cache are also stored in the cache a level higher. Three different policies exist. The first is the *inclusive* policy in which each block stored in a level of the cache is also stored in the higher level. In this type of cache when a block is retrieved from main memory, it gets stored in all cache levels. When a block is evicted from one of the levels it is also evicted from all other levels. Second is the *exclusive* policy where each block stored in the cache is not stored in the higher level. Upon an access for a line which is not stored in any of the caches, it is placed in the L1 cache. If this insertion causes an eviction from the L1 cache the evicted item gets placed in the L2 cache. The third policy is non-exclusive non-inclusive; in this policy, each block may or may not be stored in a cache level higher. For modern CPUs, the highest level cache generally is inclusive of the lower level caches [23].

Cache coherence protocols

In multi-core processors, each core can have their separate caches but caches can also be shared between multiple or all cores. When all levels of caches are shared a value written by one of the cores can directly be read by another. In modern processors, only the last level cache (LLC) is shared among all cores. A shared, inclusive last level cache improves memory sharing between cores. If a core accesses a memory address which was modified by another core the updated value must reside in the last level cache. This reduces the amount of (slower) main memory accesses required for sharing memory between CPU cores. For lines that are stored in the lower level, non-shared caches coherency problems between caches of different cores can occur. When a block resides in multiple caches, and one of the cores modifies its cached value it must be ensured that the values in the caches of other cores are updated to prevent outdated values from being used. To accomplish this, coherence protocols are used. These protocols ensure all cores see the most recent values of data when accessing it, usually by using a bus to which all caches are connected. The most widely used protocol is the MSI snooping based protocol or one of its extensions. The MSI protocol works by maintaining a state for each block in the cache. This state can either be modified (M), shared (S) or invalid (I). When a block is in the modified state, it is not consistent with the value of the block in the higher levels and must be written back upon eviction. If a block is in the shared state, it is unmodified, and the value can be used by multiple cores.

A block in the invalid state is considered not in the cache anymore. This protocol ensures that a block is only in the modified state in one cache at a time by using bus requests. For example, when a block is in the shared state and the processor issues a write request it changes to the modified state and issues a bus request to invalidate the copies of the block in other caches. To improve the cache hit ratio, several extensions to the MSI protocol have been made which add additional states such as forward and exclusive. Simulating this protocol makes cache simulation significantly more complicated. Instead of just modifying the values in the cache of the core accessing the memory, possibly the entries in all other cores have to be checked.

Instruction caches

Some processors contain dedicated caches for instruction fetches, these caches are only read memory and thus are simpler than data caches. They do not require coherency protocols and write policies, this makes their implementation easier, and therefore, they consume less physical space on the CPU.

Virtual and physical indexing

The location of a block in the cache can be based on its virtual or physical address. Because multiple virtual addresses can be mapped to the same physical address, coherency problems can occur when using a cache based on virtual addresses. However, when using a cache based on physical addresses, the address first has to be translated before the cache lookup can start. Therefore a hybrid approach can be taken. As described in Section 4.1.1, each cache entry can be identified by its tag, index and offset each of these can be based on either the physical or the virtual address. In a system with pages of size, 2^n the last n bits of the virtual address are used to determine the offset within the page. The offset is the same for the virtual and translated physical address. Thus, if the size of the cache is equal to or smaller than the size of a page, the cache index and offset will be based entirely on the page offset. In this case, there will be no difference between a physically or virtually indexed cache.

For larger caches, the tag and index can be both virtual and physical. To reduce the performance decrease caused by this translation when using virtually indexed and tagged caches the lookup in the smaller lower level caches can already be performed before the address translation has finished. To be able to support both kinds of systems the simulator should support virtual, physical and mixed indexing schemes.

4.1.2 Page tables

The page tables in a system are used to translate virtual addresses to physical addresses. Each process has its own page table, and these page tables are generally

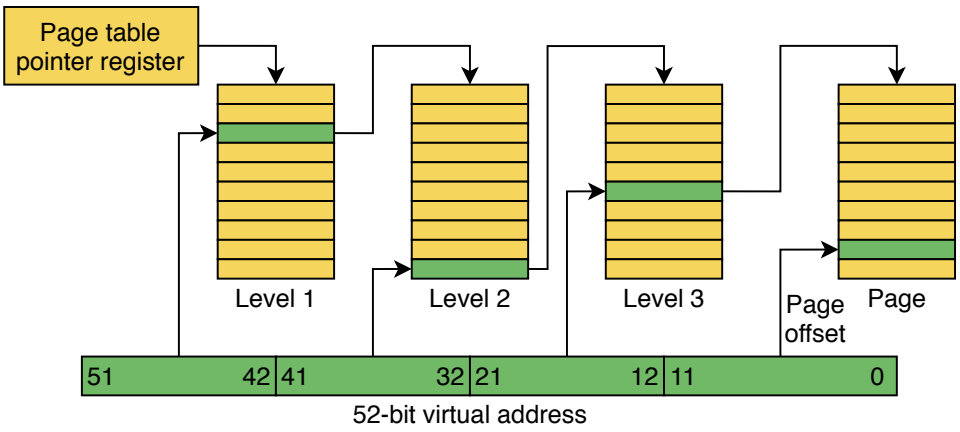


Figure 4.3: A 3-level page table for 52-bit virtual addresses with 4KiB pages

stored in the main memory of a system. If the page table containing a mapping for all possible 64-bit addresses were used, the page table alone would consume more memory than available in the system. Therefore, modern architectures like x86 and ARM use multi-level page tables. A diagram of a multi-level page table can be seen in Figure 4.3. In one of the control registers, the address of the first level of the page table can be found. The virtual address is split into multiple parts, each used as the index for one of the levels of page tables. In the first level table, the address of the second level table can be found. The entry in the second level points to the third table et cetera. Finally, the entry in the last table contains the physical address of the page together with multiple status bits. Generally up to 5-level page tables are used. To speed up the translation of virtual addresses CPUs contain a translation lookaside buffer (TLB) which maintains a cache of recently translated addresses.

4.2 Page table simulation

To simulate page tables which are stored in the guest memory, the state of the guest memory is simulated in the cache and page table simulator. The memory of the simulated system can be mirrored in the simulator by using the data traced for memory write events. When processing a traced write accesses the value at the corresponding address in the simulated memory is updated with the new value. When an address has to be translated, the page tables contained in the simulated memory can be used to find the physical address.

To be able to translate virtual to physical addresses, the values of the control registers of the CPU have to be known. The values of the control registers of the simulated CPU can be maintained in the simulator using the *guest_update_cr* event Based on these, it can be determined if paging is enabled and where the page tables to be used are located in memory.

Bit(s)	Name	Meaning
0	Present	If set the entry is valid
1	Read/Write	If not set the page is read-only
2	User/Supervisor	If set the page is a user
3	Write through	If set accesses to this page are write through
4	Cache disable	If set accesses to this page do not use the cache
5	Accessed	If set the page has been accessed
6	Dirty	If set the page has been written to
7	PSE	If set the this is a large page
8	Global	TLB entries for addresses with this bit set do not get flushed if the CR3 value is modified
9-11	Unused	
12-63	Address	

Table 4.1: An x86 page table entry

Based on the current control register values paging is enabled or disabled if paging is enabled, the simulator will use the multiple levels of the page table to find the entry in the last level. The information bits for the x86 architecture are shown in Table 4.1. These bits indicate whether the cache is disabled for the page and if the cached entries for the page are write-through. When the cache disabled bit is set for page the cache access is directly recorded in the output trace and the cache state is not updated. For a write to a page with the write-through bit set, the write should be recorded as a cache miss. The page table entry also contains a bit indicating whether the page belongs to a user process or a kernel process.

The implemented the simulator simulates an x86 style page table. However, the page table logic is decoupled from the other logic in the simulator and can thus be changed to allow simulating page table implementations of other architectures.

As previously described the guest virtual addresses have to be translated to physical addresses. This can be done using the guest page tables. Because the guest page tables reside in the main memory of the guest to be able to simulate guest page tables, the contents of the guest memory have to be known. This could be done by allocating continuous a block of memory equal to the size of the guest memory in the simulator, which is updated by traced write accesses. However, this would not allow the simulated guest memory to be larger than the amount of memory available on the machine where the cache simulation is performed. To be able to simulate page tables of the guest, only the contents of kernel owned page tables have to be known. If the kernel owns a page table can be determined using the *user/supervisor* bit in the page table entry. Considerable effort is put into reducing the size of the memory used by the kernel, therefore only allocating an array of the size of complete memory, user and kernel memory, would thus be wasteful. Instead, the simulated memory will be divided into blocks the same size as the

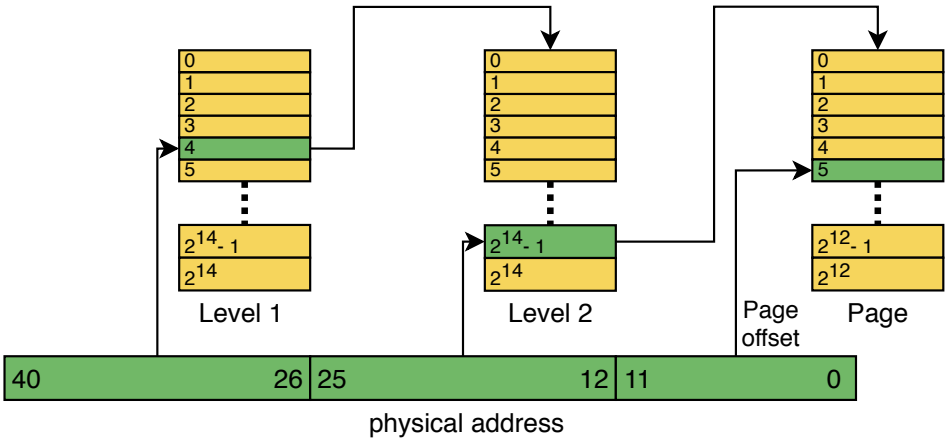


Figure 4.4: Multi level memory simulation

page size of the simulated system. These blocks are stored using a structure similar to the approach taken in multi-level page tables. For a system with a page size of 4KiB, the default page size on Linux, the address is divided into three parts, the last part has a size of 12 bits and indicates the offset within a memory block. The other 28 bits of the 40 least significant bits are divided in two 14 bit indices used search a two-level structure containing a pointer to the simulated memory block. A diagram of this approach can be seen in Figure 4.4. In total this will allow a guest memory of size 2^{40}bytes to be simulated. When an address is accessed which has not yet been allocated a new block will be allocated and a pointer added to the table structure.

Because of the added complexity, a slight decrease in performance can be expected when using the multi-level table memory simulation approach. The impact can be seen in Figure 4.5, the time it takes to simulate the first 1 billion memory accesses of three different benchmarks of the PARSEC suite using the *simmedium* input size is shown on the y-axis. The runtime for two different simulated memory sizes and the two different approaches taken is shown on the x-axis. Both the *table based* approach where a multi-level table is used and the *complete* approach where the complete guest memory is allocated in the simulator are compared. In Figure 4.6, the amount of memory used for each benchmark is shown. When the *complete* memory simulation approach is used the amount of memory used is equal to the amount of memory simulated. For the *table based* approach, the total memory usage is shown in this figure on the y-axis, on the x-axis, the amount of simulated events is shown. Notable is the fact that after approximately 1 billion simulated accesses for all three benchmarks, there is an increase in memory usage. After approximately 1 billion accesses, the setup of the benchmark by PARSEC is finished and the actual work performed in the benchmark starts. Furthermore, the influence of the working set size for each benchmark can be seen. It can be seen that freqmine has a significantly larger working set of approximately 128MiB compared

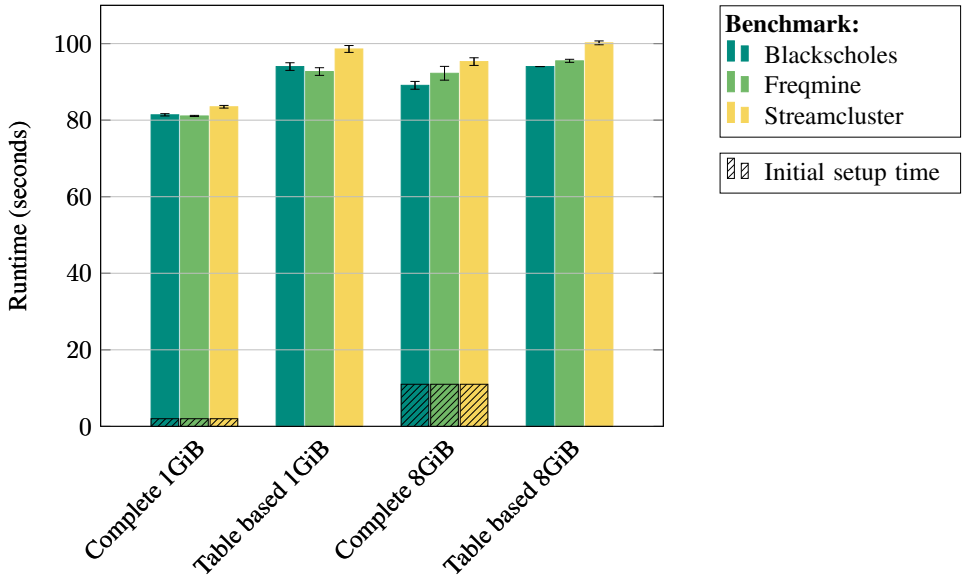


Figure 4.5: The average runtime of three executions for two different methods of simulating guest memory for the first one billion memory accesses for three benchmarks of the PARSEC suite on a host with 16GiB of RAM, the black bars indicate the standard deviation

to 16MiB for streamcluster and 2MiB for blackscholes, these values are similar to the ones described by Bienia et al. [17]. Notable is the fact that for all benchmarks amount of memory which is simulated eventually flatlines regardless of the simulated memory size. When combining the data shown in Figures 4.6 and 4.5 it can be concluded that the memory usage decreases significantly when using the table based memory at the cost of a slight increase in runtime. Therefore, the approach which is used in the page table simulator is made configurable at compile time. The user of the simulator should choose which approach to use, depending on the workload and host machine the increase in runtime might outweigh the reduced memory usage.

4.2.1 Starting page table simulation during execution

The implementation of the page table simulator described in the previous section relies on maintaining the current state of the memory of the guest from the start of the execution. Therefore tracing has to be enabled from the start of QEMU. In Table 4.2 the time to boot the operating system with tracing is enabled with */dev/null* as output together with the time it takes without tracing enabled. Because of this significant 35× slowdown, it would be beneficial to start the simulator right before the benchmark is executed after the setup has been performed. This slowdown can become even larger depending on the complexity of the setup of the benchmark.

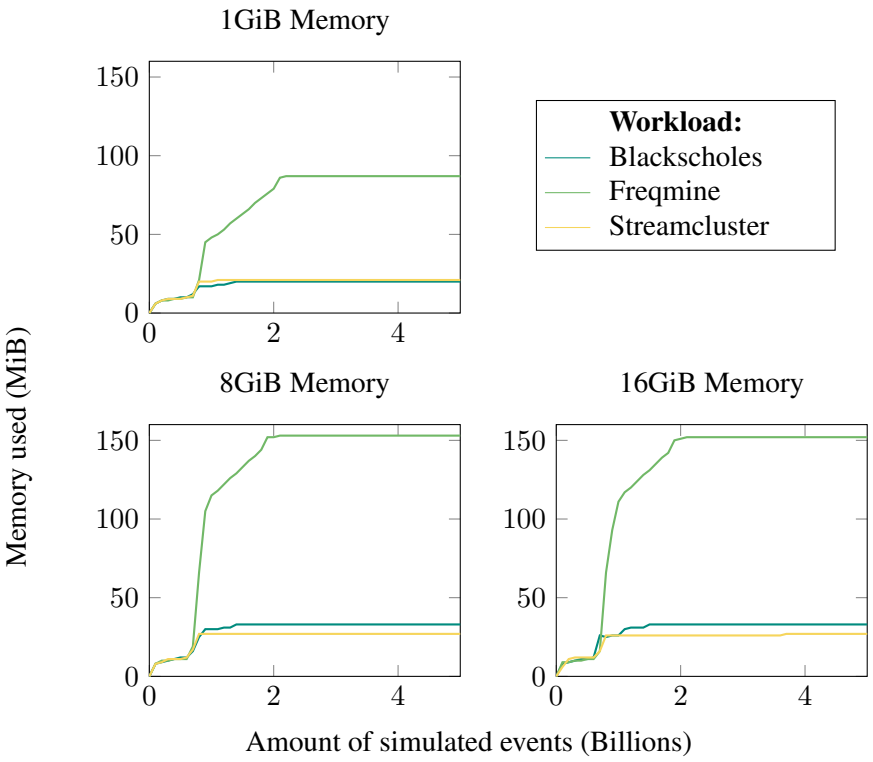


Figure 4.6: The memory simulator memory usage for different PARSEC workloads for a simulated guest system with 4 cores and various sizes of simulated memory for the first 5 billion trace events of the benchmark

Initial page table state

To be able to start page table simulation at any moment during execution, the state of the guest page tables at that moment needs to be known. The simplest approach would be to export all the page tables currently in use by the guest to a file. This file could then be read by the page table simulator to initialise contents of the memory required for page table simulation. However, to be able to export all page tables in the guest system, the memory locations of the first levels of all page tables allocated by the kernel have to be known. Preferably these values should be retrieved without modifications to the guest. This could be done by tracing all modifications to the control register containing the page table pointer (CR3 on the x86 architecture) from the start of the simulation. This way, before starting the benchmark, the simulator could be paused and the memory found at all the page table pointer values that have previously been used can be exported to a file. Unfortunately, this approach has the disadvantage that when QEMU is paused exactly when the kernel is in the process of initialising a new process, the page table for this process will not be exported. An approach which exports the complete guest memory and thus, all page tables does not have these problems.

	Runtime (seconds)	
	Average	**Standard deviation**
Tracing disabled	18.45	0.49
Tracing enabled	632.32	2.98
Slowdown	35×	

Table 4.2: The average runtime of 5 operating system boots for a simulated system with 4 cores and 8GiB of memory

Physical memory

In modern systems the available physical memory is not a continuous region, instead, several chunks are made available to the operating system. On the x86 platform, these are provided in an E820 map by the BIOS of a system, which is also simulated in QEMU. This map indicates which memory regions are usable by the operating system. To be able to determine which guest physical memory regions should be simulated by the memory simulator, this map can be used. Memory accesses to memory addresses outside these regions can also occur, like accesses to PCIe devices, for example. For memory simulation these accesses can be discarded, whether the cache should be simulated for these accesses can be determined using the caching disabled bit in their page table entry. By default, the E820 map is printed by the Linux kernel when it boots. The memory map can be extracted from this and used as input for the page table simulator.

If the all the exported guest memory regions would be loaded from the disk in their entirety when starting the memory simulator, depending on the size of the guest memory and the disk speed, this could take a significant amount of time. Furthermore, a lot of these pages might never be used. This can be solved by only loading exported guest pages from disk when they are accessed by a traced memory access.

Initial control register values

To also simulate virtual to physical address translation correctly, the initial state of the guest control registers has to be known before tracing is enabled. To interact with QEMU, two interfaces are exposed. The monitor interface is focussed on human interaction with the simulator and the QEMU machine protocol (QMP) designed to be used by machines. The QMP does not support exporting register values. However, these values can be obtained using the QEMU Monitor interface using the *info registers* command. By default, this command will print information about the values of the registers of the guest CPU. However, by default for x86, the control register values are not included. Therefore QEMU was modified to also output these control registers for the x86 host. The resulting values can be used as input for the page table simulator as the initial control register values.

Automation

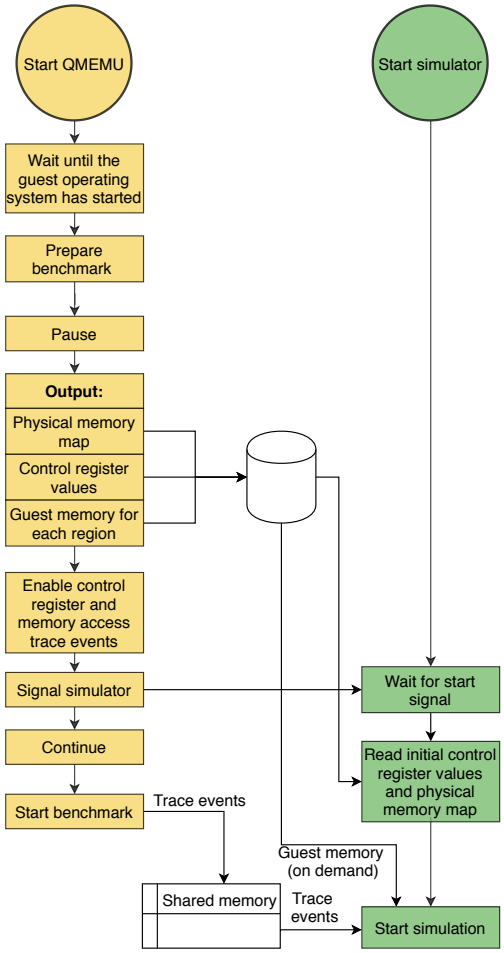


Figure 4.7: Flow for starting memory tracing for a running simulator

The complete process to start tracing can be seen in Figure 4.7. Executing the steps of this process manually each time is cumbersome. Therefore a program is developed to automate this process. This program uses a go implementation of expect [4] to interact with the guest operating system. Expect is a tool for automating interactive terminal applications; it allows waiting for particular terminal output and sending arbitrary terminal input to the guest through a pseudoterminal. These functionalities will be used to determine when the guest has booted and after it has for interacting with the host to prepare and execute the benchmark. By using expect benchmark execution on the guest can be automated without the need to modify the guest operating system.

An example of a simple expect program used to execute a benchmark can be seen in Figure 4.8. Some code to setup the pseudoterminal has been omitted for

```
...
guest := ... // pseudoterminal setup
benchmark := ` parsecmgmt -a run -p blackscholes -n 4 -i simmedium`

guest.ExpectString("ubuntu login:")
guest.ExpectString.SendLine("ubuntu")
guest.ExpectString.ExpectString("Password:")
guest.ExpectString.Sendline("Password123")

// Setup benchmark
guest.ExpectString.ExpectString("$ ")
guest.ExpectString.SendLine("cd parsec-3.0/")
guest.ExpectString.ExpectString("$ ")
guest.ExpectString.SendLine("source env.sh")
guest.ExpectString.ExpectString("$ ")

// Start benchmark
log.Println("Starting benchmark")
startTime := time.Now()
guest.ExpectString.SendLine(benchmark)
guest.ExpectString.ExpectString("$ ")

// Print time used by benchmark  (this includes the setup done by PARSEC)
log.Println("Benchmark took: ", time.Now().Sub(startTime), " seconds")
...
```

Figure 4.8: An example program for running guest benchmarks using go-expect

clarity. For exporting the physical memory map, QMP is extended with an additional *memory-info* command which outputs a list of the start address and size of all available memory regions. These regions are similar to the regions provided to the guest operating system by QEMU via the E820 map. The QMP the *stop* and *cont* commands can be used to stop and start the simulator and the *pmemsave* command is used to output the contents of a guest physical memory region to a file. Because QMP has no way to inspect CPU register values the monitor command *info registers* is used, and the human-readable output parsed.

Performance

When enabling tracing while a guest is running the time before a benchmark can be started increases from 18.45 to 40.1 seconds for a guest with 8GiB memory. This is still 16 times improvement over starting tracing at the start of QEMU compared to the 632.22 seconds (as shown in Table 4.2) before the benchmark can be started. When using this approach, the start time is independent of the complexity of the cache that is simulated. Therefore when tracing has to be started from the beginning the runtime simulating realistic, more complex, caches the improvement will become even larger.

4.3 Cache simulation

To produce a trace of memory accesses which miss the cache and are handled by the main memory of the system a cache simulator is built. This simulator can read a trace produced by QEMU from a file or shared memory. It can be configured to simulate any amount of cache levels of arbitrary size. Parameters like replacement policy used, the associativity, inclusion policy and coherency protocol can all be configured at compile time. This allows the user to evaluate the performance of a memory system for a wide range of cache configurations.

Multi-level caches can be implemented by storing a pointer to the higher level cache. When a cache is queried, but the requested line is not currently stored in the cache the higher-level cache is searched. If a cache does not have a pointer to a higher level cache and an access occurs which misses the cache the access is registered as a cache miss. Furthermore, the simulator supports separating instruction and data caches for each level. When a memory access is simulated, the correct cache will be selected based on the CPU index and whether it is an instruction fetch or data access.

4.3.1 Coherency protocols

To simulate various snooping based coherency protocols each cache also maintains a pointer to the bus it is connected to. This gives support for simulating snooping bus protocols which use a bus for message passing. These messages are seen by all other caches on the same level, which must update their state based on them. If the simulated cache is inclusive, the line should also be invalidated in all higher-level caches. If a cache miss occurs in a cache, it issues a bus request to retrieve the cache value from one of the other caches in the same level. This requires checking for each of the simulated caches on that level if it contains the entry.

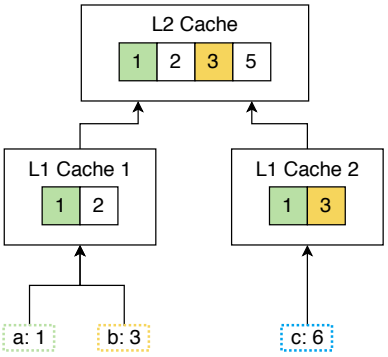


Figure 4.9: Cache inclusion for a two level inclusive cache, 3 different cache accesses are shown

When simulating an inclusive cache, this can be optimised. A visualisation of an inclusive 2 level cache hierarchy is shown in Figure 4.9. Three different accesses

are shown in this Figure:

a accesses cache line 1 in L1 Cache 1, which results in a cache hit.

b accesses cache line 3 in L1 Cache 1, which results in a miss. The line can be found in cache L2 however. This causes the cache line to be inserted in L1 cache 1, therefore, a bus request is issued to notify the other L1 caches.

c accesses cache line 6 in L1 Cache 2 which results in a miss, because the L2 cache is inclusive this cache must contain line 6 if it is stored in one of the other L1 caches. By checking the L2 cache, it can be concluded that line 6 is not stored in the other L1 cache thus no bus request has to be issued and searching the other L1 caches can be omitted.

This example illustrates an optimisation which is applied for inclusive caches. This optimisation is especially useful when simulating a cache hierarchy with a large number of caches per level or a cache with large cache sets. Depending on the coherency protocol and inclusion policy used various other optimisations similar to this one can be applied. Therefore the cache simulator is designed in a way that isolates the logic of a coherency protocol from the rest of the simulator. This allows the user to easily modify the implementation based on the inclusion policy and coherency protocol the user wishes to simulate. For evaluation of the cache simulator, the MSI and MOESI protocols are implemented.

4.4 Distributed cache simulation

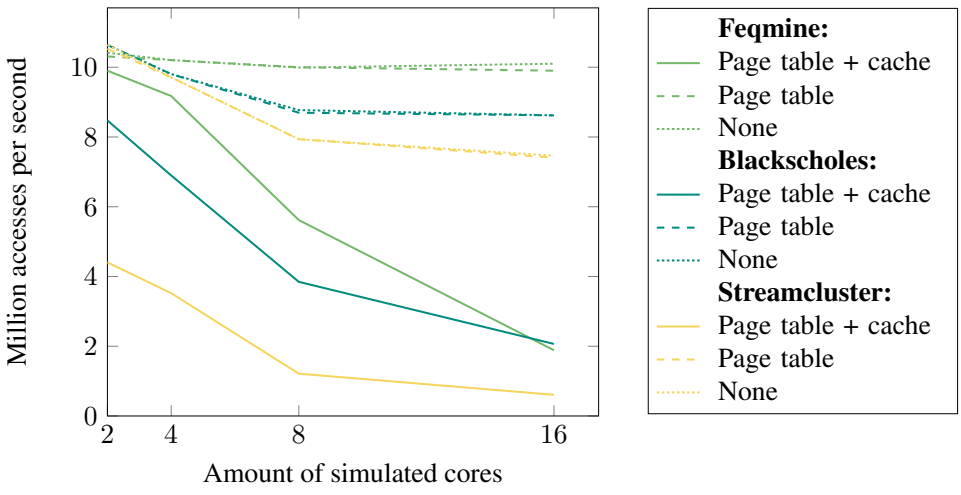


Figure 4.10: Comparison between the average amount of accesses simulated per second by the simulator with cache simulation enabled and disabled for various workloads Traces are loaded from a file

The runtime of the single-threaded cache and page table simulator described in the previous section can be seen in Figure 4.10. In this figure, the amount of simulated cores is shown on the x-axis and the average amount of events processed per second by the simulator on the y-axis. The numbers shown were gathered for multiple benchmarks using shared memory and the complete memory simulation technique. In the configuration where none of the components of the cache and page table simulator is enabled the events are simply discarded by the simulator. This can thus be regarded as the fastest QEMU can produce the events. To ensure the cache and page table simulator does not bottleneck QEMU it should be able to simulate at least as many accesses per second as QEMU produces. As can be seen from the figure when cache simulation is enabled, the performance of the simulator greatly reduces, especially for higher core counts. However, when only page table simulation is enabled, there is a significantly smaller drop in performance. Thus it can be concluded that the cache simulator is the bottleneck of the complete simulator.

The runtimes are shown in Figure 4.10, these are the runtimes when simulating a single level shared cache. When running the PARSEC blackscholes *simmedium* benchmark for a guest system with 4 cores and 8GiB of simulated memory the average runtime when simulating a cache hierarchy with a single level shared cache of 4MiB is 408 seconds. The runtime increases to 1256 seconds when simulating a realistic 3 level cache hierarchy where the L3 cache is shared, and the other two levels use the MSI coherence protocol.

Furthermore, research question 3 poses a question of how multiple different cache hierarchies can be simulated in parallel. By creating a cache simulator which can run distributed over multiple machines, the slowdown can potentially be limited, and an answer to this research question can be found. In this section, a distributed approach to a cache simulation will be proposed, since the performance impact of page table simulation is much smaller, it will not be parallelised.

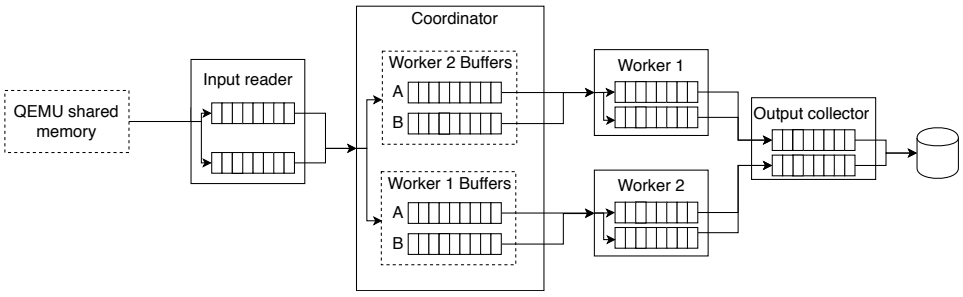


Figure 4.11: Overview of the different components in the distributed simulator

The distributed cache simulator consists of four components:

1. **Input reader**: reads the input from QEMU and sends it in batches to the coordinator

2. **Coordinator**: does memory and page table simulation and distributes translated accesses to workers
3. **Worker**: simulates one or more cache sets of a distributed cache simulator
4. **Output collector**: collects the output from the workers for each cache hierarchy

4.4.1 Input reader

When running the cache and page table simulator in a distributed manner the process reading the input, running on the same machine as QMEMU, should be as lightweight as possible. This process is called the input reader. To optimally use bandwidth, the input reader does not send each trace event individually, but it groups events in batches of trace events. These batches have the same size the two shared memory regions. The only purpose of this input reader is reading the trace output of QEMU from a shared memory region and sending this to the coordinator process. Data is sent to the coordinator using MPI [25]. For evaluation of the simulator, the OpenMPI [27] implementation is used. OpenMPI was chosen because it is widely used and is available for a wide range of platforms. The DAS-5 cluster [13] which will be used in the evaluation of the distributed simulator also supports OpenMPI.

4.4.2 Coordinator

The coordinator performs the parts of the simulator that are done sequentially for each memory access; these are:

- Control register simulation
- Memory simulation
- Page table simulation

For each memory access, the coordinator translates the virtual address and distributes cache accesses containing both addresses to different workers. To reduce communication overhead these cache accesses are buffered by the coordinator. The coordinator maintains two outbound buffers per worker. When of the outbound buffers is full, the complete buffer is asynchronously sent to the worker. Because of this asynchronous communication, when a full outbound buffer is being sent to the coordinator, it can keep processing incoming accesses by putting them in the other buffer for the worker.

4.4.3 Workers

The workers perform the actual simulation of the cache. Each worker can simulate (a part of) a different cache hierarchy. Because address translation, which

is not easily parallelizable, is independent of the cache hierarchy multiple cache hierarchies can still be simulated in parallel. This way, the after-cache trace can be gathered for many different cache types at the same time without substantially reducing the performance of the overall system. However, as seen in Figure 4.10 cache simulation is the slowest component of the simulator even when simulating a very simple cache hierarchy. Thus when cache hierarchies with high levels of associativity or complicated coherency protocols are simulated the speed of the whole system will be bottlenecked by the speed of the worker for this cache hierarchy. Therefore it should be possible to distribute simulation of a single cache hierarchy over multiple workers.

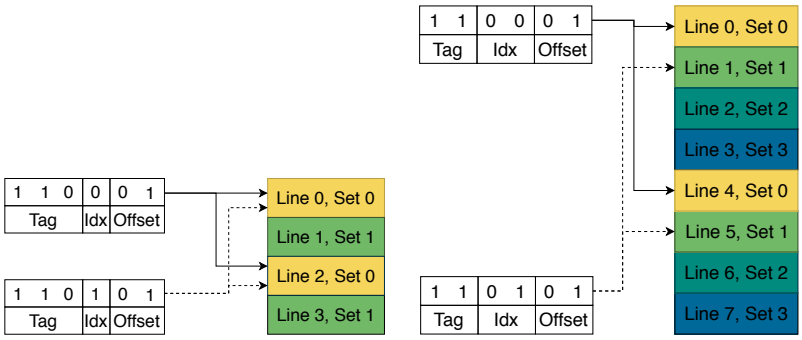
Level	Size	Associativity	Cache sets	Index bits
First level Data	32KiB	8	64	6
First level Instruction	32KiB	8	64	6
Second level	256KiB	4	1024	10
Third level	8MiB [1]	up to 16	8192 [1]	up to 13 [1]

[1] For a 4 core processor

Table 4.3: Cache specifications of an Intel Skylake microarchitecture processor [23]

Cache simulation inherently is a sequential problem. Each cache access depends on the state of the cache after the previous access has been simulated. However, when simulating a cache that is not fully associative, each cache access only depends on the previous cache accesses in the same set or equivalently: with the same index. This allows cache simulation to be parallelised, each cache set simulator can simulate one or multiple cache sets. When simulating different cache hierarchies in parallel the cache sets of more complex caches can be divided among more cache-set simulators than for simple cache hierarchies. Because cache sets can be simulated independently of each other, this will not influence the accuracy of the output trace. When simulating a multi-level cache, two addresses may be placed in separate sets of the L1 cache but get mapped to the same set of the L2 cache.

The overlapping cache sets are illustrated in Figure 4.12. Two variables influence this behaviour, the ratio between the size of a cache and the size of a higher level cache and the associativity of the two levels. When the size of a cache increases but the associativity stays the same the cache gets divided into more sets. Thus the number of index bits needed increases. When the associativity increases but the size stays the same, the cache gets divided into fewer cache sets. Thus the amount of index bits needed is less. Generally cache sizes and associativities scale with a power of 2. The amount of index bits is directly proportional to both these parameters. Therefore, if the increase in size is larger than the increase in associativity, the amount of index bits increases [23, 24]. Thus when assigning sets to workers the amount the smallest amount of cache index bits of any of the caches



(a) The 2-way associative L1 cache (b) The 2-way associative L2 cache

Figure 4.12: Two addresses that get mapped to the same cache set in the L1 cache but to different cache sets in the L2 cache

should be used to determine on which worker the access to the set should be simulated. As an example, the specifications with the required amount of index bits for a modern Intel Skylake processor are shown in Table 4.3.

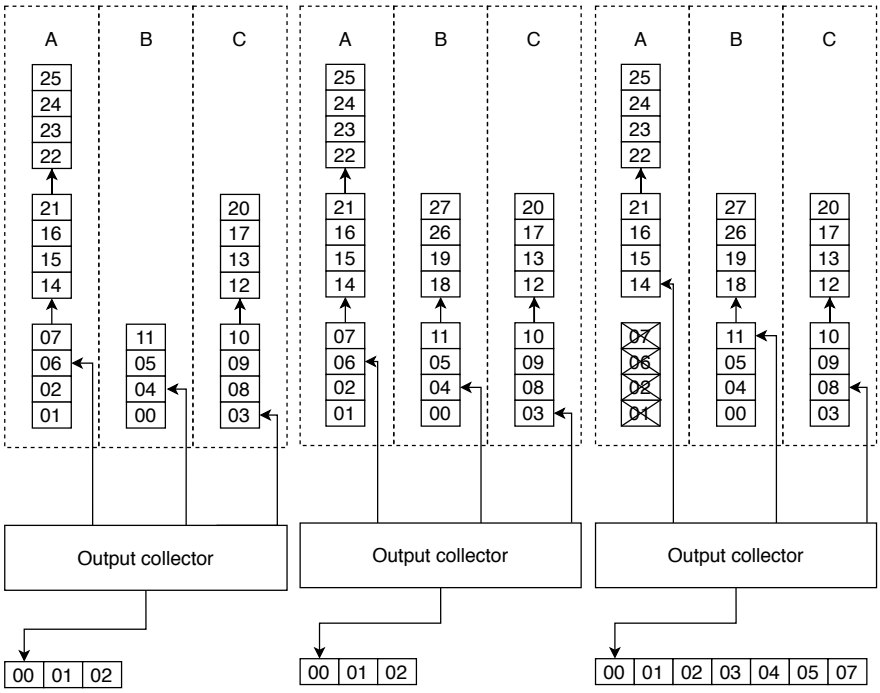
4.4.4 Output collector

When a single cache hierarchy is simulated by multiple workers, the output is scattered. For each simulated cache hierarchy, an output collector is used to solve this problem When a cache miss occurs, this is recorded in a buffer in one of the workers. Using the same two buffer approach as used in the coordinator, these buffers are sent to the output collector. The output collector will order the cache misses from different workers and write the result to a single ordered cache miss trace file. Furthermore, the collector can be used to calculate various statistics of the cache such as read to write ratio, misses per page or the total amount of cache misses.

The output collector is implemented using a linked list of buffers for each worker. This is illustrated in Figure 4.13. It will iteratively search the first buffer in the linked list of all workers for the cache miss with the smallest timestamp. When found, the cache miss is removed from the buffer and written to the output. If the entry was the last in a buffer, the buffer is freed, and the next buffer for the worker is used. This approach can handle imbalances between the amount of cache misses in different workers and can sort the output in $O(nm)$ time where n is the number of trace events and m the number of workers.

4.5 Discussion

In this chapter, the design of the cache and page table simulator was described. The design of this simulator aims to provide an answer to the sub research question 4 about closing the semantic gap. Together with the requirement posed by the tracing



(a) Before receiving a new buffer

(b) After receiving a new buffer

(c) After emptying the first buffer

Figure 4.13: The output collector

implementation in QEMU, a way to simulate the guest page tables outside QEMU was chosen. For this, all modifications to the guest kernel address space have to be traced, and the state of the memory has to be maintained in the simulator. Two approaches for memory simulation are devised, one which is more memory efficient whereas the other is designed to have a faster runtime.

Furthermore, a method is proposed to start tracing during guest execution instead of requiring tracing to be started from the guest boot. This method is shown to improve significantly the tracing time when one is only interested in tracing a specific workload.

A solution to research question 2 about how cache can efficiently be simulated is proposed. By showing how to cache simulation can be parallelised when simulating set-associative caches without influencing the accuracy of the simulator By doing this trace can be divided over multiple workers and the outputs merged to a single after-cache memory trace.

Finally, the design of a distributed system with multiple cache hierarchies simulated in parallel is presented, serving as an answer to sub research question 3.

Chapter 5

Evaluation

The design of the framework lends itself to be evaluated two parts, evaluating tracing all memory accesses and the cache simulator separately.

5.1 Experimental setup

5.1.1 Benchmarks

Traces are gathered using PARSEC which is a benchmark suite representative of real-world multithreaded workloads the [17]. The PARSEC suite provides 13 real-world programs that can be used to evaluate the performance of a system.

Besides the PARSEC suite another benchmark is used, this benchmark compresses and decompresses a file using gzip [3]. Because of the main focus on simulating multi-core systems in this thesis, a parallel implementation of gzip, called pigz [9], is used for the decompression step. The compression benchmark used in experiments compresses an 8MB generic Linux kernel using the *tar -czvf* command and unpacks it using pigz with the *tar -I pigz -xvf* command.

The benchmarks are executed using a guest system using the Ubuntu operating system version 18.04.3 LTS and version 5.0.0 of the Linux kernel. The host system used for the benchmarks has an Intel i7-4790k 4 core processor and 16GiB of memory and uses the Manjaro Linux version 19.0.0 operating system

5.1.2 DAS-5

Experiments on the performance and correctness of the distributed cache simulator are conducted on the DAS-5 cluster [13]. The DAS-5 is a compute cluster distributed over different locations, it has Intel Xeon E5-2630 v3 processors and in total has 3252 cores. Nodes are interconnected using high-speed InfiniBand links and run CentOS version 7.2.

5.1.3 gem5

To evaluate the correctness and performance of the proposed framework it is compared to the cycle-accurate simulator gem5 [19]. Gem5 is a state of the art cycle-accurate full-system simulator which is widely used to validate simulators against [20]. Memory traces from gem5 can be gathered in various places of the memory hierarchy, for example, directly between the CPU and cache or between the cache and main memory. The system simulated on gem5 uses the same operating system and Linux kernel as for the system simulated on QMEMU.

5.1.4 Metrics

The most natural metrics to compare memory access traces are the total amount of reads, writes and fetches for a given benchmark. For workloads in the PARSEC benchmark suite, some of these values are known [17] and can be used to analyse the accuracy of the simulator.

To evaluate the accuracy of cache simulators the metric misses per thousand instructions (MPKI) can be used [20]. The traces used in this thesis contain up to tens of billions of accesses. Therefore, a modified version of this metric is proposed: misses per ten million instructions (MPTMI).

5.2 QMEMU tracing

5.2.1 User mode tracing

To evaluate the accuracy of traces gathered using QMEMU, these can be compared traces obtained using CMP$im and gem5. For the PARSEC benchmarks with the simlarge input the amount of read and write accesses and the cache miss ratio are known, measured using CMP$im [17]. These numbers only include the accesses performed by the benchmark, not those performed by the operating system. When QMEMU is configured to run in user mode using the TCG traces for of memory accesses performed by only the benchmark can be obtained. This comparison can be used to verify the correctness of the TCG memory tracing implementation.

Setup

To obtain these numbers, QMEMU is configured to run in user mode using TCG. The *guest_mem_load_before_exec* and *guest_mem_store_before_exec* trace events are enabled. The benchmarks are executed using 8 threads. Equal to the number of threads used to obtain the numbers presented by Bienia et al. [17]. For the benchmarks, the simlarge input set is used. The precompiled binaries available from the PARSEC website [10] are used.

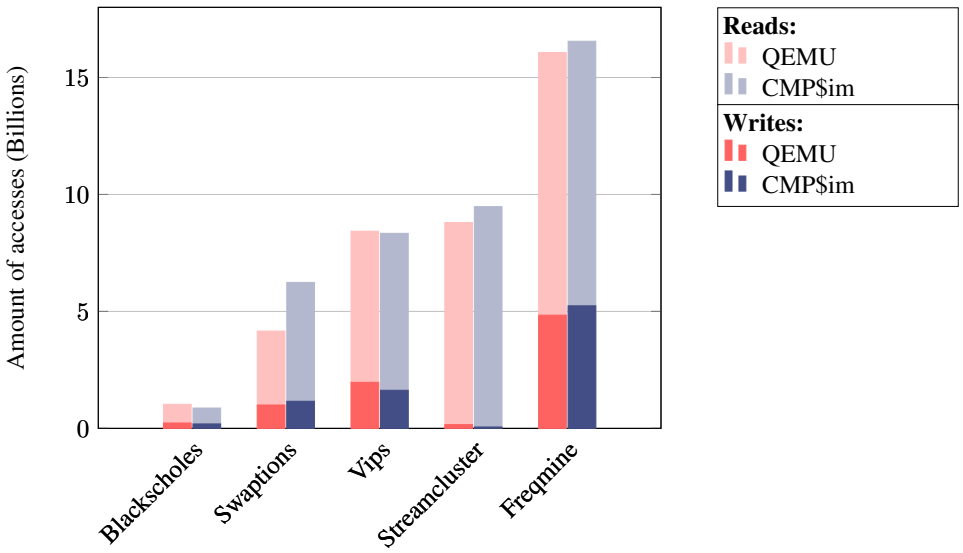


Figure 5.1: Analysis of traces produced using QMEMU user mode tracing and gathered using CMP$im for the simlarge benchmark inputs [17]

Results

The results of running the benchmarks in QMEMU user mode can be seen in Figure 5.1. In this figure, the amount of read and write memory accesses obtained using QMEMU and CMP$im for 3 benchmarks of the PARSEC suite are shown. CMP$im can reach a simulation speed approximately 4-10 million instructions per second (MIPS) [33]. Whereas the average instructions reached by QMEMU for user-mode tracing range from 47 to 61 MIPS for the used benchmarks. These values were calculated based on the measured runtime and the instruction counts for used benchmarks as described by Biena et al. [18]. The average difference in the amount of traced accesses for the different benchmarks between QMEMU and CMP$im is 7.1%. Keeping the significant increase in MIPS in mind that can be handled by QMEMU 7.1% can this increase in accuracy could be considered a worthwhile tradeoff.

5.2.2 Full system

To evaluate the accuracy and performance of full system tracing the traces will be compared against traces gathered using the cycle-accurate gem5 simulator.

Setup

Gem5 provides an example full system configuration which can be used for the benchmarks. Multiple benchmarks are executed using this example system, during

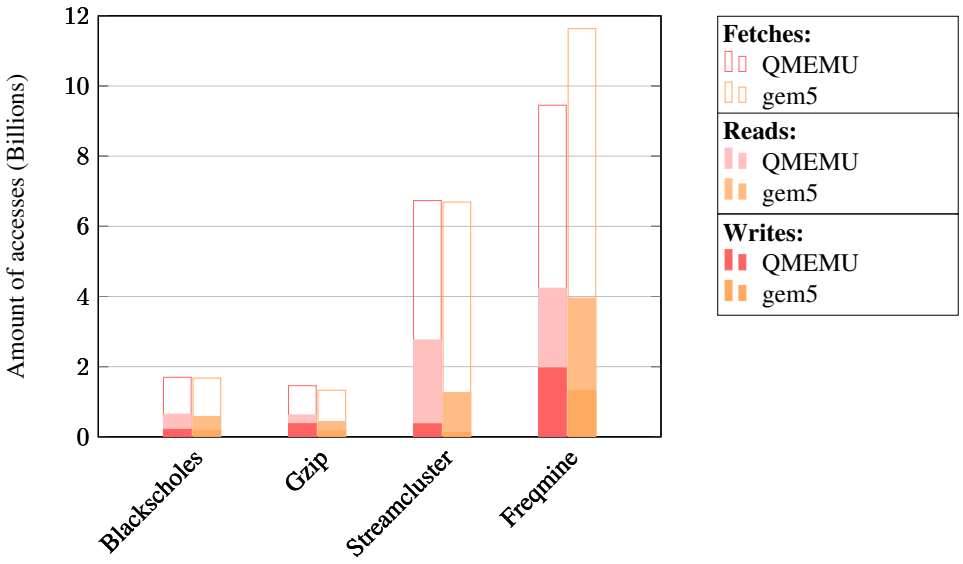


Figure 5.2: The amount of memory accesses traced when tracing a full system for gem5 and QMEMU and various workloads

which memory access traces gathered using MemTraceProbes that store traces to the disk. In gem5 multiple different CPU models with different accuracies can be used to collect these traces, the AtomicSimpleCPU is used, this is the fastest, but regarding timing the least accurate CPU model available. The example script is modified by placing a CommMonitor between the CPU and memory to collect traces of all memory accesses performed by the simulated CPU.

Traces are also collected using QMEMU. The traces collected using gem5 contain physical addresses, whereas the traces collected using QMEMU contain virtual addresses. Therefore, the virtual addresses in the traces collected using QMEMU will be translated to physical addresses using the page table simulator. Traces from both sources are analysed and compared based on various metrics. Because of the significant runtime of the gem5 simulator the *simmedium* input size is used for the PARSEC benchmarks instead of the *simlarge* input.

5.2.3 Results

In Table 5.1, a comparison between the output gathered using QMEMU, and the output gathered using gem5 can be seen. Because of the high runtime when running large multithreaded benchmarks on gem5, it is only feasible to simulate a single core system running the PARSEC *simmedium* benchmark size. A difference in read/write ratio between QMEMU and the gem5 tracing solutions, QMEMU consistently has a lower read/write ratio than gem5. Apart from the streamcluster benchmark, the amount of write accesses traced using QMEMU is less than when gem5 is used. Contrarily, the difference in the total amount of similar. This could

Benchmark	Simulator	Read/write ratio	Runtime (seconds)	Speedup
Blackscholes	gem5	2.23	2913	
(simmedium)	QMEMU	2.06	110	26.5
Streamcluster	gem5	9.20	9489	
(simmedium)	QMEMU	6.44	223	42.6
Freqmine	gem5	2.00	17915	
(simmedium)	QMEMU	1.16	1083	16.5
Gzip	gem5	2.64	2392	
	QMEMU	1.66	194	12.3

Table 5.1: Analysis of traces produced using QMEMU tracing and gem5 for a simulated guest with 1 core and 8 GiB memory

be caused by some memory accesses which are classified as an instruction fetch by QMEMU but classified as read by gem5.

A significant difference in runtime can be seen, the largest runtime difference is measured for the streamcluster benchmark where gem5 is 43 times slower than QMEMU.

5.3 Page table simulation

To verify the correctness of the page table simulation approach, QMEMU was modified to trace both the virtual and physical address for each memory address. This was done using the approach described in Section 3.5. Using the virtual addresses of these traces page tables where simulated. By utilising these simulated page tables the virtual address of each access was translated to the corresponding physical address. This was done for the PARSEC blackscholes and freqmine simlarge benchmarks. Each benchmark was executed twice, once when tracing was started from the start of QMEMU and a second time where tracing is enabled during execution just before the benchmark was started. Using a simple program to analyse the traces it was verified that all physical addresses acquired using the page table simulator are equal to the ones traced by QMEMU. Thus when assuming the correctness of the QMEMU MMU, it can be concluded that the page table simulation approach works correctly.

5.4 Cache simulation

5.4.1 Accuracy

The cache simulator can be compared to gem5 to evaluate its accuracy. Gem5 is a widely used tool to compare cache simulators against [20]. As previously

Level	Size	Associativity	Shared	Coherence	Inclusive
L1 Data	32KiB	8	no	MOESI	-
L1 Instruction	32KiB	8	no	MOESI	-
L2	256KiB	4	yes	-	Mostly

Table 5.2: Cache specifications of the simulated cache

described, memory probes can be placed in arbitrary places in a cache hierarchy in gem5. For the evaluation of the cache simulator, these probes are placed in two different places in the cache simulated by gem5. First of all, probes are placed between the simulated CPU and L1 caches. Furthermore, a probe is also placed between the L2 cache and memory bus. Each memory trace probe generates a separate trace file. These files are merged into a single trace file using a memory trace analyser program developed especially for this purpose. This program also converts the merged traces from the gem5 protobuf format to the format used by the cache simulator. A trace probe will also be placed between the L2 cache and main memory to produce after-cache traces. To verify correct behaviour a realistic cache hierarchy will be simulated, and the results for gem5 and the proposed cache simulator can be compared using a plot of the MPTMI for both simulators. The properties of this cache are similar to those of an Intel Skylake processor. The exact values can be found in Table 5.2.

By default the gem5 memory system simulates the MOESI protocol and has mostly inclusive caches, therefore the proposed cache simulator will also use the MOESI protocol but it will simulate fully inclusive caches. The trace used is a trace of the boot of the operating system followed by the setup and execution of the blackscholes benchmark with the *simmedium* input size for a 4 core system.

The amount of MPTMI for both simulators for the first 5 billion events of the execution of the blackscholes simlarge benchmark is shown in Figure 5.3. In this graph, the total amount of simulated accesses is shown on the x-axis, the MPTMI are shown on the y-axis. It can be seen that the amount of cache misses found when using the custom cache simulator closely follows the amount of cache misses found by gem5. A small difference between gem5 and the proposed cache simulator can be expected because gem5 simulates a ”mostly” inclusive inclusion policy which is not implemented for the proposed simulator. The cache simulator is configured to maintain a strict inclusion policy for this experiment.

5.4.2 Parallel cache set simulation

The DAS-5 system is used to measure the speedup gained by running the cache simulator in a parallel manner. For this experiment, the parallel cache simulator runs on a single node. On this single node, multiple cache set simulators run in parallel, each simulates a subset of all cache sets. A trace file of the PARSEC blackscholes *simlarge* benchmark containing the first five billion memory accesses

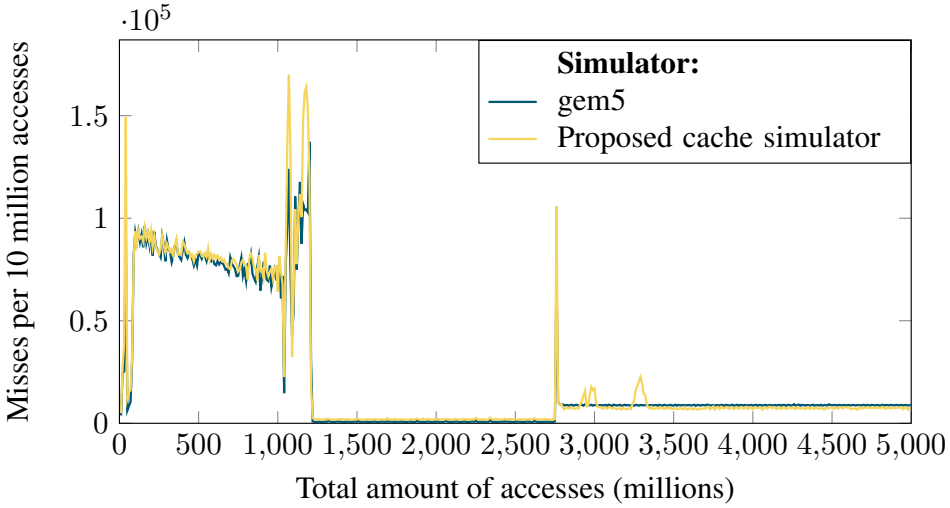


Figure 5.3: Misses per ten million accesses for the first 5 billion trace events of a trace of a the execution of blackscholes on gem5

is used as input. The cache simulates has 3 levels of which the last level is shared, the sizes of these levels are respectively 32KiB, 256KiB and 8MiB. In Figure 5.4, the average number of memory accesses that are processed per second can be seen on the y-axis for a range of 1 to 7 cache set simulators running in parallel. It can be seen that there is a slight superlinear speedup. The simulated cache uses more memory to simulate the guest cache than available in the cache of the host system. This is an explanation for the slight superlinear speedup when divided into more sets a larger portion of the simulated cache can fit entirely in the cache of the host which improves performance.

5.4.3 Distributed cache hierarchy simulation

To evaluate the distributed simulator, a setup similar to the one used for evaluating the parallel cache set simulator is used. However, for this experiment, multiple cache hierarchies are simulated at the same time. The same realistic cache hierarchy, as described in Table 4.3 is simulated multiple times in parallel. Each hierarchy is simulated on a separate node and uses 8 parallel cache set simulators, from Figure it can be seen that this should be sufficient to reach the maximum simulation speed. The throughput is shown when OpenMPI is configured to use conventional ethernet and when the high-performance InfiniBand interface is used. These interfaces have respectively, a maximum throughput of 1GiB/s and 48Gbit/s. In the leftmost graph, the throughput of the single-threaded cache simulator can be seen. The performance of this simulator is similar to the performance of the distributed implementation when using ethernet. When InfiniBand is used, the distributed simulator is approximately 8 times as fast as the single-threaded simulator. This

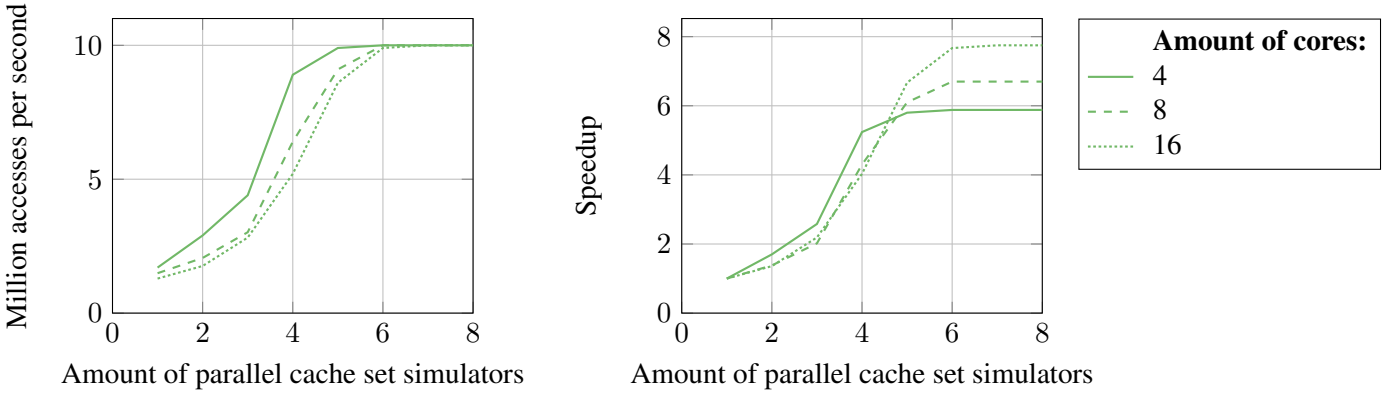


Figure 5.4: Runtime for the parallel simulator, the average million accesses per second that are processed are shown when processing the first billion accesses of the blackscholes simlarge benchmark

is a significant improvement. It can be seen that when using InfiniBand, the performance reduces slightly for each additional cache added. However, when using the slower ethernet, the performance halves when simulating changing from 1 to 2 or from 2 to 3 simulated caches. Thus the performance decrease in the distributed simulator can mainly be attributed be to the communication overhead in the coordinator.

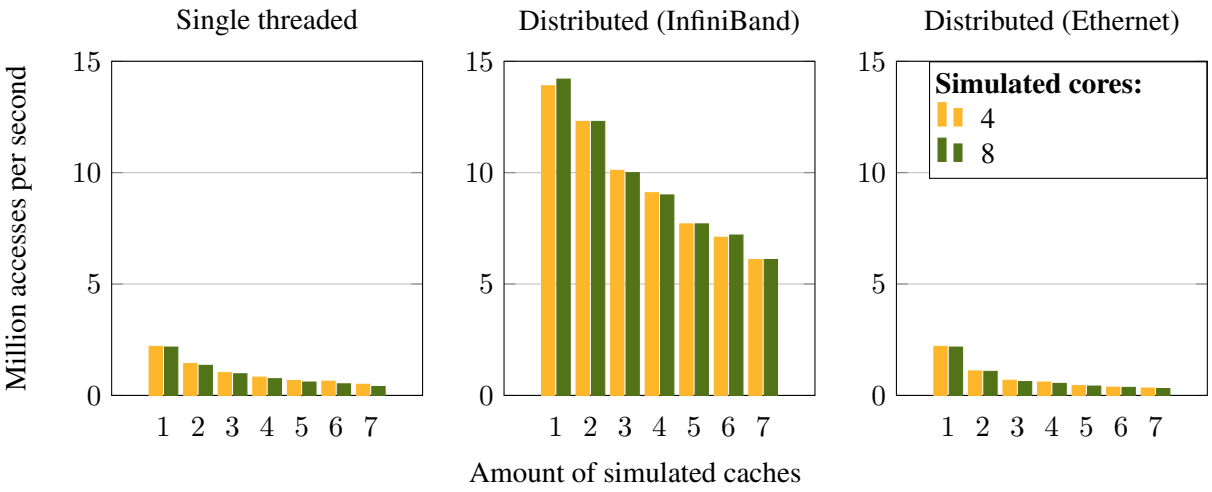


Figure 5.5: The performance of the distributed cache simulator when simulating multiple caches simultaneously (traces are read from a file)

5.5 Discussion

From the results of the experiments in this subsection, it can be concluded that the accuracy of the traces generated using QMEMU approaches the accuracy of those generated using gem5. There is a difference in accuracy between the read/write ratios traced using QMEMU and gem5, however. Furthermore, it is shown that collecting traces using QMEMU is more than an order of magnitude faster than collecting traces using gem5.

From the experiments, it can be concluded that the page table simulator produces a correct translation from virtual physical addresses. Furthermore, from Figure 5.3, it can be seen that the misses recorded by the cache simulator closely resemble those found by gem5 for the same workload. Therefore the proposed cache and page table simulator serves as an answer to sub research question 2 on how before-cache traces can be converted to after-cache traces. Being able to simulate the guest page tables outside QMEMU correctly allows the user to calculate various statistics using these, thereby providing an answer to research question 4.

Chapter 6

Conclusions and recommendations

6.1 Conclusions

This thesis is centred around the research question:

How can full-system after-cache memory access traces be gathered?

This question was divided into four sub-questions. By designing a framework and implementing it for the x86 architecture these questions have been answered.

1. *How can full-system memory access traces be gathered with minimal performance impact?*
   Using the proposed implementation it is shown that it is feasible to gather full-system before cache memory access traces using a modified version of the widely used QEMU full system simulator. This can be done by leveraging the tracing functionality of QEMU and extending the TCG generator to allow tracing instruction fetches. The approach of using QEMU is shown to have a significant performance improvement over other full system tracing solutions like gem5 without a large difference in accuracy. For the PARSEC benchmark suite, a speedup up to 42.6 times is achieved over gem5. It was found that collecting memory traces containing physical addresses is not possible using any known simulator without a significant performance penalty.

2. *How can multi-core CPU caches be simulated efficiently to convert before-cache traces to after-cache traces?*
   To be able to simulate caches in an efficient manner a design for a parallel functional cache simulator is proposed and implemented. This simulator receives a trace of memory accesses with their virtual address from QMEMU.

To translate these to the physical addresses which are required for accurate cache simulation the page table simulator maintains the state of the guest memory and control registers. This allows simulating the guest page tables from the traces. These physical addresses can then be used in a configurable cache simulator to simulate the caches of the guest and generate after-cache traces. Cache simulation is shown to be compute-intensive when simulating multi-level caches with advanced coherency protocols. Therefore a way to parallelise it is proposed, this parallelisation is achieved by dividing the memory access trace by the cache set to which each memory accesses belongs. This parallelisation does not influence the accuracy of the simulator. It is shown that this results in a significant speedup which allows the cache simulator to process 10 Million accesses per seconds. This speedup allows the cache simulator to keep up with the speed at which QMEMU generates traces.

3. *How can different types of CPU caches be simulated in parallel?*
   The proposed and implemented system is able to simulate different cache hierarchies in parallel using OpenMPI. A single coordinator performs the control register and page table simulation to translate the virtual to physical addresses and distributes the memory accesses over multiple workers. These workers each simulate a different cache hierarchy which allows the user to investigate the influence of different cache hierarchies on memory usage. It is shown that the performance penalty of simulating the multiple caches in parallel this way is significantly lower than when using a sequential approach. The approach allows 7 caches to be simulated concurrently at the cost of only a two-fold increase in simulator runtime.

4. *How can page tables be traced to reduce the semantic gap?*
   Because the answer to subquestion 1 already dictates that the page tables are simulated outside QEMU they can easily be traced. Simulating the guest tables outside QEMU allows the user of the framework to collect various metrics on the guest page tables. Furthermore, the taken approach allows the after cache traces to contain both the virtual and physical address for each memory access, no other known simulator is able to produce these kinds of traces. These metrics can be collected without requiring additional modifications to QEMU or the guest operating system and without having an impact on the performance of the system. These collected metrics can be used to help close the semantic gap. For example, by knowing from which process each memory access originated, the number of page tables each process used or whether an access was caused by the operating system kernel or a user process.

By combining the different parts of the framework used to answer the sub research questions an answer to the main research question of this thesis can be formulated in the form of the complete framework. It is shown that the proposed

framework is able to gather full-system after-cache traces with an accuracy similar to state of the art cycle-accurate gem5 simulator. This proposed simulator can collect these traces up to 42.6 times faster than state of the art solutions like CMP$im and gem5. This performance increase makes it possible to generate traces for real-world benchmarks whereas previously traces could only be collected for artificial workloads. Furthermore, the proposed framework is able to provide additional information about the guest. This information can help close the semantic gap and allows simulating cache hierarchies with complex coherency protocols. Because the page table simulation is performed in the stand-alone cache and page table simulator it is able to simulate caches with a mixed virtual and physical indexing scheme. No other publicly available cache simulator is able to do mixed addressing according to Brais et al. [20]

The source code of the systems developed for this thesis is publicly available [1]. Because of the size of memory traces used during experimentation is in the order of hundreds of gigabytes traces are not distributed. The traces can be re-generated using the tools used which are publicly available.

6.2 Discussion and future work

In this thesis, a solution for full-system after-cache memory tracing is searched. The purpose of being able to collect these kinds of traces is to gain a better understanding of memory usage in modern systems. The design of a framework that can collect these traces is devised in this work. The framework consists of two parts, a before-cache memory tracing solution QMEMU and a distributed page table and cache simulator.

It was shown that QMEMU is significantly faster than widely used tools like CMP$im and gem5. Unfortunately, this improvement in speed comes at the cost of a reduced accuracy when gathered traces are compared to those collected by gem5. Because of the loosely coupled design of the proposed framework, QEMU could easily be replaced by another source of before cache traces if the user requires a higher accuracy. QMEMU is implemented in an instruction set architecture-independent manner but the implemented page table simulator is focussed on the x86 platform. However, regardless of this dependency many of the techniques applied to build this simulator can be applied to other architectures. The page table simulator allows the cache simulator to simulate mixed indexed caches and helps close the semantic gap between raw memory access traces and their meaning. Furthermore, a way to simulate direct-mapped or set-associative caches was proposed. This simulator allows exploration of different cache hierarchies in parallel at an accuracy coming close to the accuracy provided by cycle-accurate simulators.

Like all other research, there is always room for extension. As described the improved performance of QMEMU comes at the cost of reduced accuracy of the

[1] `https://github.com/doriandekoning/full-system-memory-tracing-overview`

collected traces. This work primarily focused on being able to correctly produce complete memory access traces of the simulated guest. In the future, more work could be done to improve the accuracy of these traces. To achieve a higher accuracy the powerful QEMU TCG could be modified further to improve accuracy.

Currently, the QEMU guest MMU is used to trace memory accesses made by the guest for translating virtual to physical addresses. Unfortunately, the TLB implemented in QEMU is purely functional and does not resemble an actual TLB. By simulating a more accurate TLB the overall accuracy of the traces could also be improved.

Besides improving QEMU trace accuracy future work could instead also explore the abilities to reduce the runtime of gem5 and allow it to work with the cache and page table simulator proposed in this paper. There has already been an attempt to parallelise gem5 in the past, unfortunately, this attempt was unsuccessful.

Because the proposed framework allows tracing realistic workloads, it paves a way for research into new metrics and methodologies to analyse memory behaviour of modern systems with new memory topologies.

Bibliography

[1] Dinero IV trace-driven uniprocessor cache simulator n. http://pages.cs.wisc.edu/~markhill/DineroIV/.

[2] Gartner forecasts worldwide public cloud revenue to grow 17.5 percent in 2019. https://www.gartner.com/en/newsroom/press-releases/2019-04-02-gartner-forecasts-worldwide-public-cloud-revenue-to-g.

[3] GNU gzip compression utility. https://www.gzip.org/.

[4] Go expect. https://github.com/Netflix/go-expect.

[5] Intel cpi rate. https://software.intel.com/en-us/vtune-help-cpi-rate.

[6] Marssx86 performance summary. http://www.marss86.org/~marss86/index.php/Marss_Performance_Summary.

[7] Marssx86 running into problem with linux kernel 4.3. "https://www.mail-archive.com/marss86-devel@cs.binghamton.edu/msg02284.html".

[8] Memory access analysis for cache misses and high bandwidth issues. https://software.intel.com/en-us/vtune-help-memory-access-analysis.

[9] Parallel implementation of gzip (pigz). https://zlib.net/pigz/.

[10] Parsec benchmark downloads. https://parsec.cs.princeton.edu/download.htm.

[11] perf: Linux profiling with performance counters.

[12] ARM. Programmers guide for ARMv8-A (DEN0024A) version 1.0, 2015.

[13] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer*, 49(5):54–63, 2016.

[14] Yungang Bao, Jinyong Zhang, Yan Zhu, Dan Tang, Yuan Ruan, Mingyu Chen, and Jianping Fan. Hmtt: A hybrid hardware/software tracing system for bridging memory trace's semantic gap. *arXiv preprint arXiv:1106.2568*, 2011.

[15] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. 2007.

[16] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[17] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[18] 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. ACM, 2008.

[19] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[20] Hadi Brais, Rajshekar Kalayappan, and Preeti Ranjan Panda. A survey of cache simulators. *ACM Computing Surveys (CSUR)*, 53(1):1–32, 2020.

[21] Carlos Carvalho. The gap between processor and memory speeds. In *Proc. of IEEE International Conference on Control and Automation*, 2002.

[22] Eric S. Chung, Michael K. Papamichael, Eriko Nurvitadhi, James C. Hoe, Ken Mai, and Babak Falsafi. Protoflex: Towards scalable, full-system multiprocessor simulations using fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, 2(2):15:1–15:32, June 2009.

[23] Intel Coorporation. Intel 64 and IA-32 architectures optimization reference manual, 2016.

[24] Advanced Micro Devices. Software optimization guide for AMD family 17h processors", 2017.

[25] Message Passing Interface Forum. *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015*. High-Performance Computing Center Stuttgart, University of Stuttgart, 2015.

[26] Y. Fu and D. Wentzlaff. Prime: A parallel and distributed simulator for thousand-core chips. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 116–125, March 2014.

[27] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 97–104. Springer, 2004.

[28] K Grimsrud, J Archibald, M Ripley, K Flanagan, and B Nelson. Bach: A hardware monitor for tracing microprocessor-based systems. *Microprocessors and Microsystems*, 17(8):443–459, 1993.

[29] Wim Heirman, Trevor Carlson, and Lieven Eeckhout. Sniper: Scalable and accurate parallel multi-core simulation. In *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES-2012)*, pages 91–94. High-Performance and Embedded Architecture and Compilation Network of , 2012.

[30] Takahiro Hirofuchi and Ryousei Takano. Raminate: Hypervisor-based virtualization for hybrid main memory systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 112–125. ACM, 2016.

[31] Yongbing Huang, Licheng Chen, Zehan Cui, Yuan Ruan, Yungang Bao, Mingyu Chen, and Ninghui Sun. Hmtt: A hybrid hardware/software tracing system for bridging the dram access trace's semantic gap. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(1):1–25, 2014.

[32] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276. ACM, 2009.

[33] Aamer Jaleel, Robert S Cohn, Chi-Keung Luk, and Bruce Jacob. Cmp $im: A pin-based on-the-fly multi-core cache simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA*, pages 28–36, 2008.

[34] Sooyong Kang, Sungmin Park, Hoyoung Jung, Hyoki Shim, and Jaehyuk Cha. Performance trade-offs in using nvram write buffer for flash memory-based storage devices. *IEEE Transactions on Computers*, 58(6):744–758, 2008.

[35] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*, page 2. ACM, 2007.

[36] 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. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[37] Rano Mal and Yul Chu. A flexible multi-core functional cache simulator (fm-sim). In *Proceedings of the Summer Simulation Multi-Conference*, page 29. Society for Computer Simulation International, 2017.

[38] H. Matsuo, S. Imafuku, K. Ohno, and H. Nakashima. Shaman: a distributed simulator for shared memory multiprocessors. In *Proceedings. 10th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, pages 347–355, Oct 2002.

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

[40] VM Oracle. Virtualbox, 2015.

[41] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. MARSSx86: A full-system Simulator for x86 CPUs. In *Design Automation Conference 2011 (DAC'11)*, 2011.

[42] Avadh Patel, Furat Afram, and Kanad Ghose. Marss-x86: A qemu-based microarchitectural and systems simulator for x86 multicore processors. In *1st International Qemu Users Forum*, pages 29–30, 2011.

[43] Jan S Rellermeyer, Maher Amer, Richard Smutzer, and Karthick Rajamani. Container density improvements with dynamic memory extension using nand flash. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, page 10. ACM, 2018.

[44] Marc Rittinghaus, Thorsten Groeninger, and Frank Bellosa. Simutrace: A toolkit for full-system memory tracing. *White paper, Karlsruhe Institute of Technology (KIT), Operating Systems Group*, 52, 2015.

[45] Josef Schneider, Jorgen Peddersen, and Sri Parameswaran. A scorchingly fast fpga-based precise l1 lru cache simulator. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 412–417. IEEE, 2014.

[46] Thiem Van Chu. *Ultra-Fast and Accurate Simulation for Large-Scale Many-Core Processors*. PhD thesis, Tokyo Institute of Technology, 2015.

[47] Matt T Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 23–34. IEEE, 2007.