Acceleration of memory accesses in dynamic binary translation

Antoine Faravelon

▶ To cite this version:

Antoine Faravelon. Acceleration of memory accesses in dynamic binary translation. Operating Systems [cs.OS]. Université Grenoble Alpes, 2018. English. NNT : 2018GREAM050 . tel-02004524

HAL Id: tel-02004524
https://theses.hal.science/tel-02004524

Submitted on 1 Feb 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Communauté UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Antoine FARAVELON

Thèse dirigée par **Frédéric PETROT (MSTII)**, Professeur Grenoble-INP, Grenoble INP
et codirigée par **Olivier GRUBER**

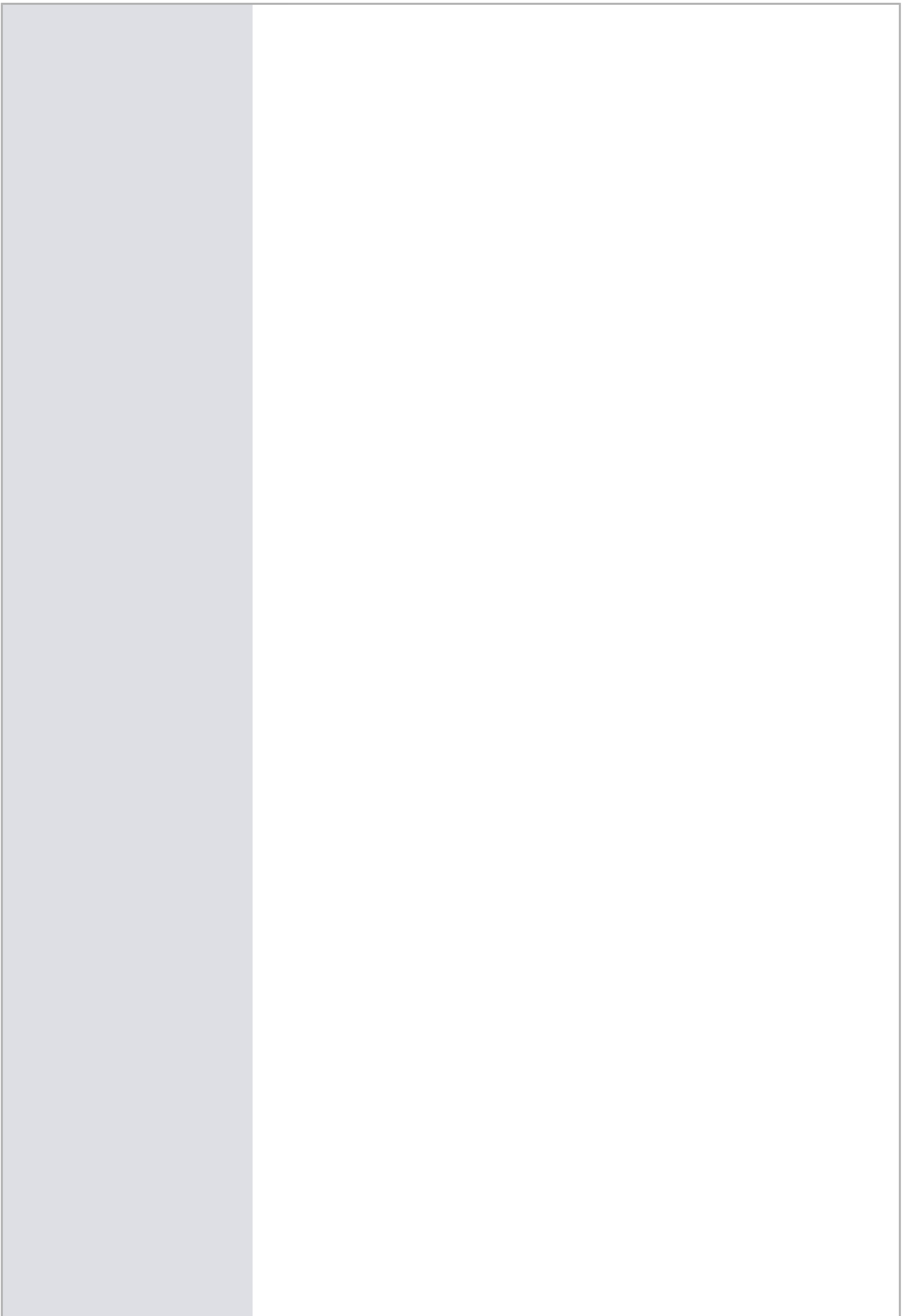
préparée au sein du **Laboratoire Techniques de l'Informatique et de la Microélectronique pour l'Architecture des systèmes intégrés**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Accélération des accès mémoire dans la traduction binaire dynamique

Acceleration of memory accesses in dynamic binary translation

Thèse soutenue publiquement le **22 octobre 2018**,
devant le jury composé de :

Monsieur FREDERIC PETROT
PROFESSEUR, GRENOBLE INP, Directeur de thèse
Monsieur ERVEN ROHOU
DIRECTEUR DE RECHERCHE, INRIA CENTRE RENNES-BRETAGNE ATLANTIQUE, Rapporteur
Monsieur GAËL THOMAS
PROFESSEUR, TELECOM SUDPARIS, Rapporteur
Madame FLORENCE MARANINCHI
PROFESSEUR, GRENOBLE INP, Président
Monsieur ALAIN TCHANA
MAITRE DE CONFERENCES, INP TOULOUSE - ENSEEIHT, Examinateur
Monsieur OLIVIER GRUBER
PROFESSEUR, UNIVERSITE GRENOBLE ALPES, Co-directeur de thèse



Remerciements

Je tiens tout d'abord à remercier mon directeur de thèse Frédéric Pétrot. Malgré son emploi du temps très chargé du à ses responsabilités, il a toujours été disponible pour discuter et m'accompagner tout du long de cette aventure. Et ce bien que je n'ai pas été le doctorant le plus simple à encadrer (on se contentera de citer ma tendance au travail nocturne). Je remercie aussi Olivier Gruber d'avoir co-encadré et largement participé à l'élaboration de cette thèse. Mais aussi d'avoir su me garder motivé avec sa version bien à lui de *l'optimisme*. Je remercie bien sûr Nicolas Fournel qui, non content de m'avoir donné goût pour les architectures matérielles, est également celui qui m'a permis de rejoindre le laboratoire TIMA pour mon magistère, ouvrant ainsi la voie à cette thèse.

Je remercie ensuite mon Jury de thèse. Tout d'abord Florence Maraninchi pour avoir accepté de présidé ce jury. Erven Rohou et Gaël Thomas pour leur retours très intéressants et très positifs. Et aussi Alain Tchana pour avoir examiné cette thèse, et pour ses remarques et perspectives très intéressantes.

De manière plus étendue, je remercie tout ceux qui m'ont accompagné durant ces trois ans, ceux de SLS et de GreenSocs: Luc et Clément qui ont subis mes assauts de questions, Adrien P., Thomas et Olivier M. pour toutes les discussions que nous avons pu avoir, techniques ou non et avec lesquelles ont peut toujours refaire le monde, Adrien B. avec qui j'ai fais mes armes sur Qemu. À Martial et Arief qui ont vécu avec moi les derniers mois de thèses, préparation à l'oral et de rédaction. Et bien sûr un grand merci à Enzo qui a été un excellent public pour les répétitions de ma présentation. Finalement un grand merci à Laurence, avec laquelle l'enseignement à l'UGA et la fin de ma thèse ont été largement plus amusants !

Et je n'oublie pas l'essentiel, en remerciant ma femme, Danmei, que j'ai rencontré aussi dans l'équipe SLS. Merci de m'avoir donné l'envie de continuer et de m'avoir soutenu à tout moments. Sans elle, cette thèse n'aurai sans doute pas connu la même réussite.

Contents

1 Introduction **1**

2 Problem statement **3**
- 2.1 Instruction Set Simulation Techniques 3
- 2.2 Full system simulation with DBT 6
  - 2.2.1 Dynamic Binary Translation principle 6
  - 2.2.2 Simulating complex instructions 8
  - 2.2.3 Emulating memory accesses 9
- 2.3 Conclusion 13

3 State of the art **15**
- 3.1 Dynamic Binary Translation 15
- 3.2 Accelerating Full System Dynamic Binary Translation 17
  - 3.2.1 Generated code optimizations techniques 18
  - 3.2.2 Multi Threading support 20
- 3.3 Accelerating Memory Accesses Simulation 22
  - 3.3.1 TLB based solutions 27
- 3.4 Conclusion and Remaining questions 28

4 Accelerating DBT using hardware assisted virtualisation for cross-ISA simulation **31**
- 4.1 Introduction 31
- 4.2 Background 32
  - 4.2.1 Hardware Assisted Virtualization 32
  - 4.2.2 Dune framework 34
- 4.3 Design 36
- 4.4 Implementation 38
- 4.5 Conclusion 42

5 Cross-ISA simulation using a Linux kernel module and host page tables **45**
- 5.1 Introduction 45
- 5.2 Design of our Solution 46
- 5.3 QEMU/Linux Implementation 49
  - 5.3.1 Kernel module 50
  - 5.3.2 Dynamic binary translation 53
- 5.4 Handling Internal Faults at Kernel Level 56
- 5.5 Conclusion 58

6 Experimentations **61**
6.1 Introduction 61
6.1.1 Benchmark set 61
6.2 Breakdown of QEMU execution time 62
6.2.1 Ratio of time spent in subparts of QEMU 62
6.2.2 Instruction breakdown: percentage of memory accesses in the benchmark 64
6.2.3 Efficiency of existing memory accesses simulation acceleration 65
6.3 Dune based solution 66
6.3.1 Protocol 66
6.3.2 Performance Analysis 67
6.3.3 Communication costs 69
6.4 Linux Kernel Module solution 70
6.4.1 Performance Overview 71
6.4.2 Hybrid solution analysis 72
6.4.3 Address space mapping analysis 73
6.4.4 Performances with in kernel fault handling 76
6.5 Conclusion and comparison of both methods 78

7 Conclusion **81**
7.1 Prospective 83
7.1.1 N privilege levels support 83
7.1.2 QEMU Multi threading support 85
7.1.3 64-bit targets support 85
7.1.4 Handling write accesses with in kernel internal faults handling 87

Chapter 1

Introduction

Computing systems are now ubiquitous in our daily life, be it under the form of a computer, or that of a device that performs computations unbeknownst to its user. These systems have taken a major place in modern society thanks to silicon integration that makes them smaller and smaller, and yet more and more feature full, leading to the advent of System on Chip (SoC), now and for some time already, the dominant way of selling silicon.

The integration trend makes the design and programming of these chips more and more complicated. It is indeed nowadays fairly common for the ones that are relatively "general purpose" to feature 8 or more cores, gigabytes of RAM and advanced graphics units. To make matters worst, chip makers are producing dozens of different chips each year. Due to this, neither designers, which need to test their designs, nor developers which need their applications to be available as soon as possible for the new chips, can wait for the final tape out. A partial solution to these problems is the use of simulation. Instead of waiting for the chip to be available, a software model of it is conceived, and executed on top of a general purpose computer. Amongst the models, one is of particular interest to us, the model of the processor that executes the code of the operating system and applications.

To model instruction set architectures, multiple techniques have been developed throughout time. Historically, the first and simplest one has been instruction per instruction interpretation. However, it has been shown to be very slow. To avoid this problem, more sophisticated techniques were developed. Amongst them we can find dynamic binary translation, static binary translation and native (or compiled) simulation. Although the fastest, static binary translation, and native simulation even more so, can not directly run any unmodified target code we might want.

On the other hand, dynamic binary translation is fairly balanced. It is relatively fast (from four to twenty times slower than native execution) and precise enough to run unmodified target software. It is at the base of many industrial grade simulators and in particular of QEMU, which is one of the most widely used simulators nowadays. QEMU itself is used in industrial grade products such as Android Emulator, it is also the official KVM frontend.

While dynamic binary translation already provides satisfying performances, it sill has multiple aspects that could be improved. Out of those, one important bottleneck is the memory subsystem. Each and every target memory access actually has to go through multiple intermediary steps to be executed. In particular, if the architecture we wish to simulate, called the target, uses virtual memory, the target memory management unit will have to be simulated. The result will be called a software MMU. As it is done in software, this simulation will automatically be much slower than target hardware.

In this thesis, we present solutions to improve memory accesses simulation speed in the

context of cross-ISA dynamic binary translation. More precisely, we choose to implement our solution in QEMU. To do so, we offload the process of managing target memory accesses to host hardware, so as to obtain near native performance. This has been done through two methods, one based on HAV, and one based on a companion Linux kernel module for QEMU. In both cases, the goal is to give QEMU access to a fully controllable address space, which in turn can be used to provide the target with its own native like address space.

Our best performing solution currently yield a 78% speedup over baseline on average, with speedups of up to 6 times over baseline. It can manage multiple QEMU targets, and runs full fledged OSes such a Linux. Very few modifications have been made to QEMU and the Linux module is fairly small and maintainable.

The manuscript is organized as follow. Chapter 2 details precisely the problems we identified. Chapter 3 explores solutions which have already been given to these problems, and details what we consider is left to be done. Chapters 4 and 5 present the solutions we designed and developed to address the remaining problems. In Chapter 6 we present the performance analysis we made, and use it to compare the two solutions we propose between each others, and with already existing solutions available in the literature. Finally, Chapter 7 concludes this thesis, and draws some research perspectives.

Chapter 2

Problem statement

In this chapter, we shall present the main problems faced by fast CPU simulation in the context of full system instruction set simulation (ISS). In particular, we shall talk about the main technologies used to face this problem and their limitations. We will more particularly focus on the limitations of dynamic binary translation (DBT), currently the most efficient and also probably the most widespread amongst these technologies. We will finally detail the problem behind the core of this work, simulating memory accesses in full system cross-ISA simulation using DBT.

2.1 Instruction Set Simulation Techniques

In this section we will cover the most common instruction set simulation techniques that exist today. These techniques rely on translating the instruction set of a given architecture, called the target, onto that of another one on which the code will be executed, the host. Multiple techniques exist to reach that goal, each of those have their advantages and drawbacks. However, they tend to follow a simple rule: the faster the technique is, the less accurate it is. While accuracy, in term of estimating the time it takes to execute some program on the target for example, is not always an important concern, abstracting away too many details may leads to also losing the ability to simulate any type of code. We will now present those ordered from the slowest to the fastest in term of simulation speed, and thus, consequently, also from the most to the least accurate.

Interpretation

Interpretation is probably the oldest and the most accurate simulation technique.

The process is as follows. First we read the current instruction, whose address is given by the CPU program counter register, from the target memory model. Then this instruction is decoded so as to extract the type of operation to perform, its inputs, its outputs and its possible side effects. Once this is done, a corresponding piece of code modelling this operation will be executed. This principle is outlined in Algorithm 1. We first retrieve the instruction from the mem array which represents the memory. We then decode it, and two cases are presented. Either it is a simple register to register operation like the "ADD", in which case we simply read all source registers concerned by the operation, effectuate the computation and store the result in the destination register. Or it is a memory access, as illustrated by the "LDR" case, and then we first read the src register that contains the virtual

while *simulation is running* **do**
  $instruction \leftarrow mem[pc]$
  $dec \leftarrow decode(instruction)$
  switch $dec.opcode$ **do**
    ...
    case ADD **do**
      $regs[dec.dest] \leftarrow regs[dec.src1] + regs[dec.src2]$
    end
    ...
    case LDR **do**
      $phys_addr \leftarrow softmmu_translate(regs[dec.src])$
      $regs[dec.dest] \leftarrow mem[phys_addr]$
    end
    ...
  end
end

Algorithm 1: Typical interpretation based simulator algorithm

address at which the access should be performed. We then translate it, if needed, to a physical address by making a target page walk. And finally we store the result in the destination register.

The process has the advantage of being fairly accurate, as it progresses instruction after instruction. In term of simulation, it can even be cycle accurate. It is also able to run any type of target code, without modifications. Another big advantage is that it is fully portable, *i.e.* it does not depend on the host. A fairly popular i386/x64 simulator, often used for OS development, Bochs [MS08], still uses this process nowadays.

Its main disadvantage is that its speed is relatively limited. Works from [RBMD03, PGH+11] shows that each instruction execution is at least one hundred times slower than native code execution. Some optimizations do exist, the main one, called *predecoding*, being to store instructions in their decoded form in some cache (usually the model of the instruction cache) and directly reusing this decoded form when the same program counter is reached and the cache hits [TGN95].

Dynamic binary translation

Dynamic binary translation, as its name implies, consists of dynamically translating target binary code into host binary code. Phases of simulation will then be interlaced with phases of code generation. It could in many ways be seen as a direct optimization of interpretation, where the result of decoding target instruction, instead of only being executed, is translated into host code and kept for later reuse. Its main advantage is that it remains relatively accurate, and is able to execute any type of target code, while being orders of magnitudes faster than raw interpretation. A precise description of this technique is given in section 2.2.

Static Binary Translation

Static binary translation (SBT), as opposed to dynamic binary translation, aims at translating the whole target binary before actually executing the resulting code [CVERL02]. The main advantage of this technique is that since the translation is done offline, no side effects are

seen by the user at runtime. For example, in DBT, some latency might be seen at program launch, or even on user input due to the need of generating code dynamically. Also, more time on optimizing the translated code than in DBT may be spent. Its main disadvantage is that translating code when branches targets are not known statically is quite complex, and in quite a few situations even impossible. Simulating self modifying code, or dynamically loading executable code is also tedious. This usually induces the need of having, embedded in the generated code, an interpreter which, at runtime, interprets offending code. Simulating OS code also requires OS paravirtualization to detect the new code and interpret it. At the end of the day, this means that running unmodified target binaries is not always possible.

Native Simulation

Native (or compiled) simulation is probably the fastest cross ISA simulation technique nowadays. At the difference of binary translation techniques, it relies on the target source code. This source code will be compiled directly for the host, with potentially code annotation to be able to reconstruct the execution on real target machine. The necessary accesses to devices will be interfaced with a Hardware Abstraction Layer (HAL), which somehow is a paravirtualization layer. The backend of the HAL can then interface with the device model, written in plain C or SystemC for example. The main limitations of this technique is that first the simulator needs the source code of the application, thus disqualifying proprietary and legacy code, which is often unacceptable. The second disadvantage of this method is that it is highly imprecise. Almost none of the target machine execution information may be retrieved without code instrumentation. The third and technically most restrictive limitation is that it can not easily be used to simulate an OS. In its standard form, this technique indeed relies on knowing in advance everything that will be executed and compile it so as to be able to execute it on the host. This makes it unadapted for full system simulation where no knowledge exists on what will be simulated.

Hardware assisted virtualization

Hardware assisted virtualization (HAV), although not directly targeted at simulation, can still be used to fill some of its roles [SHP12]. Its base principle is to partially duplicate the host CPU so that an unmodified target may run on the duplicated CPU. This duplicated CPU, called vCPU (for virtual CPU) has its own architected state, its own address space as well as its own view of every architectural register. With this method, the target can run at near native speed, unmodified and with the possibility for the host to monitor it. It can be used for example to develop a new OS, or debug an existing one, as well as for deeper performance evaluations with tools such as `perf` [DM10]. Its only real limit is that both the target and host must have the same ISA. This defeats the purpose of testing new CPU designs, and running legacy or in general binaries from other ISAs.

Summary

To summarize, multiple CPU simulation techniques exist, however only one of them is currently truly able to fit the needs of full system simulation while remaining relatively fast : dynamic binary translation. Other techniques, such as native simulation, may indeed be intrinsically faster, but they are severely limited in term of full system simulation. At best, they need heavy paravirtualization techniques to function, and are usually limited to applications. Hardware assisted virtualization could be used for same ISA simulation,

but can not, at least directly, be used for cross ISA simulation [HPF13]. As such, it seems reasonable to focus on dynamic binary translation for our goal of accelerating full system simulation.

2.2 Full system simulation with DBT

Let us now present how full system simulation using DBT functions. We will detail its current limitations, with a particular emphasis on what interests us the most in this work, the simulation of memory accesses.

2.2.1 Dynamic Binary Translation principle

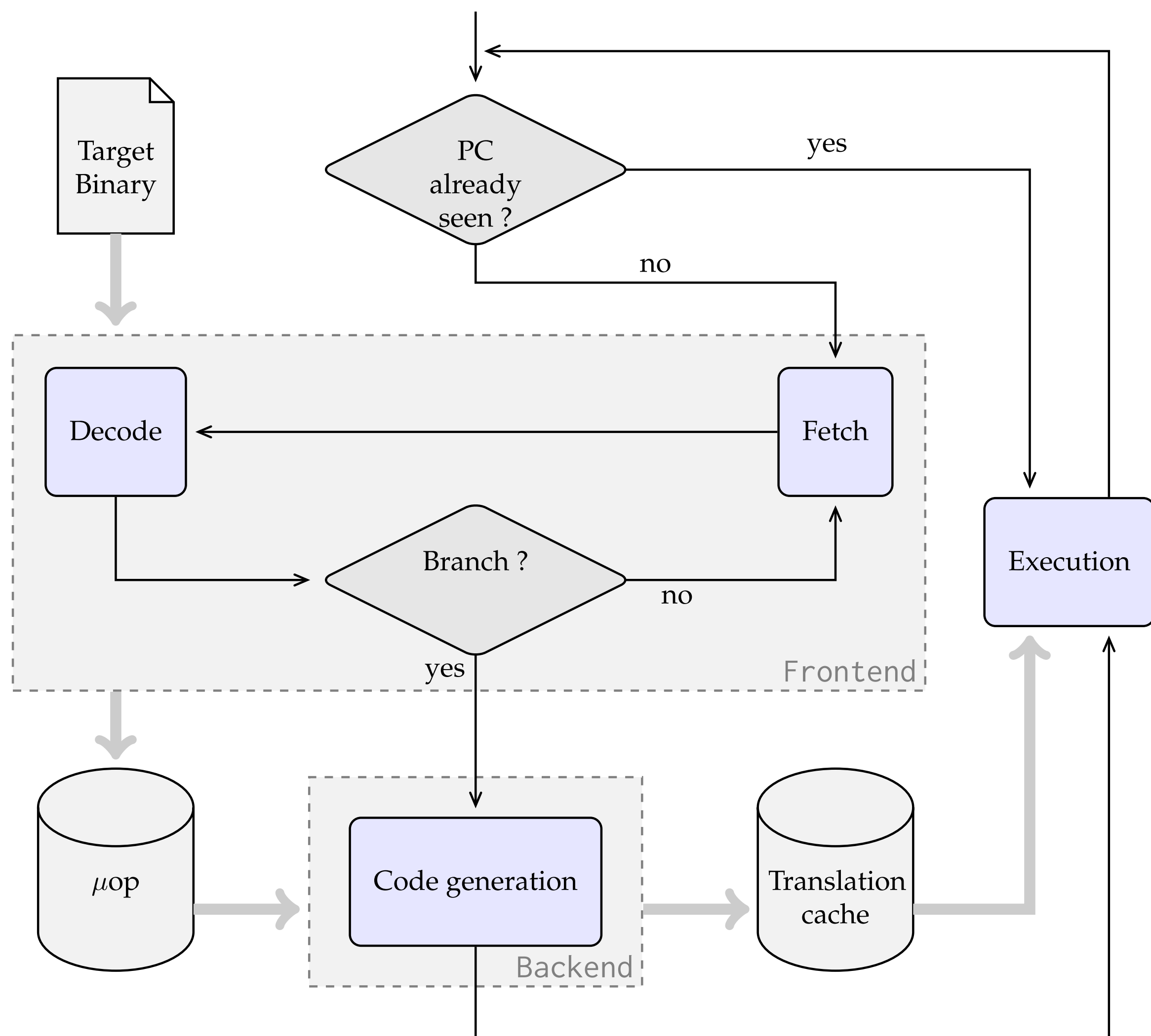


Figure 2.1: Overview of the DBT process
(taken and translated from [Mic14])

Let us first explain the base principles of retargetable dynamic binary translation. The DBT based simulator fetches, instruction per instruction, the code of the target program. This code is translated dynamically to first an intermediate representation (IR) used by the

simulator to decorrelate the target ISA from the host ISA. Then this IR is translated to host binary code. Finally, this code is executed directly on the host. A more precise view of the way DBT works is depicted in Figure 2.1. The process begins at the fetch phase. There, we fetch the target instruction located at the address corresponding to its current program counter (PC). Then, this instruction is decoded *i.e.* the target opcode is translated to a corresponding micro operation that is part of the IR used by the simulator. As long as the instruction is not a branch, we go back to the first step to fetch and translate the next instruction. When finally the currently decoded instruction is a branch, all the micro operations are translated to host binary code. This code is then cached into a translation cache containing translations for the code that has already been encountered. The block that is thus stored is called a translation block, and it is fairly similar in spirit to a basic block as defined in compilation. Finally, the code in the translation block is executed. When arriving at the end of the block, the simulator checks whether the program counter (PC) at which the last branch instruction points has already been seen. If yes, the execution will resume at the thus pointed translation block. Otherwise, it will go back to the fetch-decode phase.

An important point to remember is that the code cache size is not unlimited. Eventually, as we fill it with more and more code, it will be full. At this time, a choice will have to be made so as to evict some already translated code blocks to make room for new ones. A simple yet surprisingly efficient choice is to flush to whole translation cache.

The main advantage of this process is that it does not make any assumption on the target binary. Thus, whether we are simulating an operating system or application code, whether the code is self modifying or if it dynamically loads libraries or executables, nothing changes for the simulator, the fetch-decode-execute process continues. Thanks to using an IR, it is also usually fairly retargetable, *i.e.* it can support many (target, host) ISA couples. The simulation speed is also vastly superior to interpretation as has been shown in [CK94, WR96, AEGS01].

One limit of full system DBT is that the generated code is usually not much optimized. The reason for that comes from the nature of this code. In full system simulation, one OS and multiple processes will be simulated. At runtime, we have no informations on what we are executing, the result is that many pieces of code we generate might never be reused. Code will also tend to get modified or suppressed by the target resulting in translation blocks flushes. Jumps will happen in multiple places in a block, thus making multiple relatively small partial copies of it. The result is that the opportunities for optimization are limited, and the chances that an advanced optimization be really useful is limited. Moreover, advanced optimizations are time-consuming. Spending too much time optimizing will result in spending a long time to generate the translation blocks, and if the blocks are not reused in the end, this cost will not even be amortized. This issue is dealt with in great detail in the work from [DB00].

The way the code is generated, *i.e.* instruction per instruction and block by block, will also have an important impact on performance. Usually, existing optimizations will not concern more than a translation block at a time. This means that advanced techniques inspired from other just in time (JIT) compilers such as dynamic inlining and code specialization will be fairly rare in full system DBT. The code will also be marked by the need of keeping the architected state up to date, and in particular target registers. This means that for most instructions, a phase that loads the registers state will often precede the actual simulation code, which will then be followed by a phase that stores the modified registers. The resulting code for a target (Arm in this case) instruction translated to host (x64 here) can be seen in Figure 2.2. We can see that the target instruction, which made no memory accesses to begin with, has been converted to not a single one, but three instructions, with two of them being memory accesses. The first instruction loads the value of the target `ip` register into a host

Target source code

```
add ip, ip, #20  ; 0x14
```

Generated Code

```
0 : mov 0x30(%r14), %ebp   ;Obtain current target register value
1 : add 0x14, %ebp         ;Execute the computation
2 : mov %ebp, 0x30(%r14)   ;Write back the result to target register
```

Figure 2.2: translation of a target add instruction

register. The second executes the actual computation. And the third one stores the new value of the register into the architected state. Just from sight, we can imagine the resulting code will be much slower than native code. The host `add` will probably execute in one cycle, but both the load and store may need several cycles to execute. And this example is for a simple instruction, complex instructions might result in much more host code to simulate them.

2.2.2 Simulating complex instructions

Another performance bottleneck of DBT is the simulation of complex instructions. What we call complex instructions are those that can not directly be translated from a target instruction to a near equivalent host instruction. Those come from three groups.

- The first group is architected state updates. Those are instructions that change the state of the CPU and not just of a standard architectural register. This could be a TLB flush instruction, or a switch to a different privilege level. These usually imply an important amount of checks and modifications to be done by the simulator and would be highly impractical to do directly in IR,
- The second group are the vectorized and floating points instruction sets. Those instructions are usually fairly hard to map from one ISA to another as compliance to the standard and legacy are conflicting interest in floating-point unit designs. An extensive analysis of floating-point simulation can be found in [SBP16]. Regarding integer vectorized instruction, [MFP11] has proposed extensions of an IR to include SIMD instructions and demonstrated large speedups on applications extensively using these instructions.
- The final group is memory accesses. Those heavily depend on the architected state of the target, and in particular of its MMU for target virtual address to target physical addresses translation. Due to the code implied to make a target page walk, C code is almost unavoidable in current simulators. We will explain memory accesses emulation more precisely in 2.2.3.

From now on, we focus on memory accesses. The main solution to simulate those instructions is to call a C function, called a helper, that emulates the target instruction behaviour. Such a call is depicted in Figure 2.3. As we can see, a helper call induces an important overhead compared to standard generated code execution. It will induce at least an argument setup phase (we place arguments for the helper in the right registers or at the

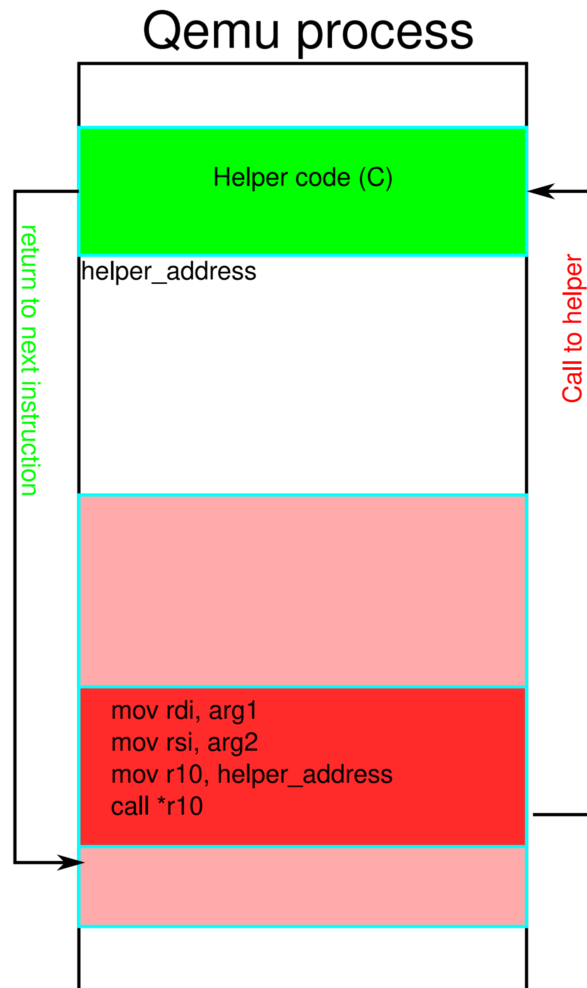


Figure 2.3: Call of a C code helper

appropriate position on the stack), a jump to the helper, register saving then restoring due to the call, and an indirect jump back to generated code. Just these phases, without simulation code, is already much more than say a memory access or a standard vector instruction in term of execution time. This makes helpers an important element of slowdown compared to the raw execution of generated code.

2.2.3 Emulating memory accesses

We will now detail more precisely the techniques currently used to simulate memory accesses. Emulating memory accesses in the context of DBT is actually fairly complex. This task consists in making an unmodified target binary believe that it executes in an address space that is the same as in real hardware. This implies simulating multiple key points of the target machine, including its MMU, all its memory instructions and memory related faults. All of this also brings a fairly large architected state, with the notions of privilege levels, page size, potentially page table format.

For the sake of clarity, in this section, we will consider that the target ISA is based on a 32-bit ARM processor with an MMU and that the host ISA is an Intel x64 machine.

From the guest perspective, everything happens as if the emulator was not there. Guest

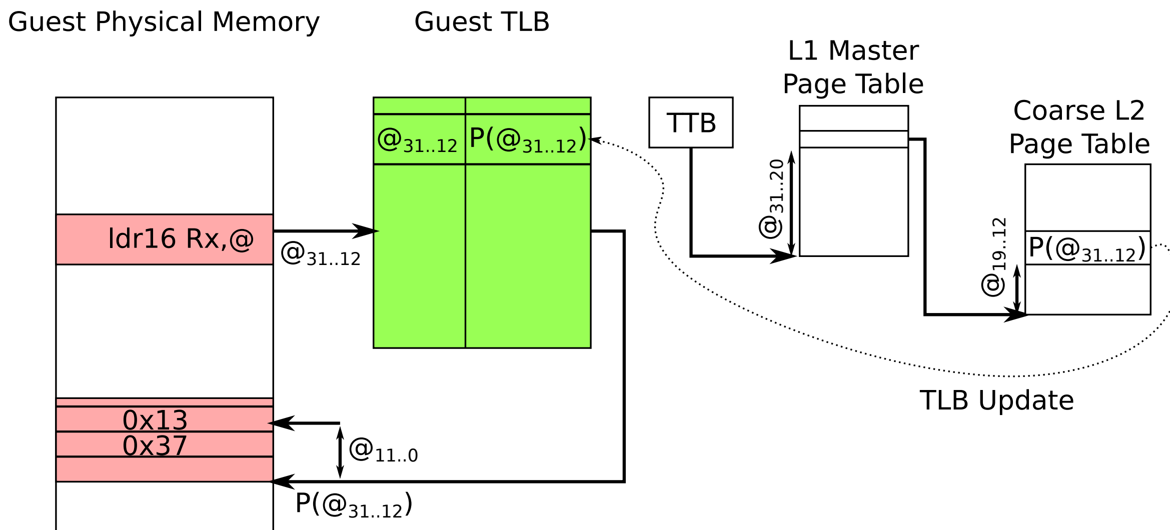


Figure 2.4: Simplified view of the hardware support to address translation on the ARMv7 architecture

assembly instructions execute as expected on the guest hardware. So for instance, the ARM instructions for memory accesses, `ldr` and `str`, executes as expected, either manipulating physical or virtual addresses. If virtual, this means that the MMU has been set up and enabled. This guest perspective is depicted Figure 2.4, looking at a `ldr` instruction. In this picture, we assume pages of 4KB (the smallest granularity for ARMv7). We classically call virtual page number (VPN) the page number obtained by bit 32 to 12 of the virtual address, and physical frame number (PFN) the translation of these 20 bits into physically mapped pages. When executing the load, the hardware accesses the *Translation Lookaside Buffer* (TLB), a small fully-associative memory that caches the most recent virtual to physical translations, to obtain the page frame number, as depicted on the left-hand side of the figure. If the address translation is not cached (TLB miss), the page table structure that contains the OS defined VPN to PFN mappings for the current process (identified by the Translation Table Base register, TTB) is traversed. If the translation is found, the TLB is updated and the instruction finishes. If not, a page fault exception is raised, and it must be handled by the OS running on the guest.

On Figure 2.5, we take the host perspective, showing the *architected state* maintained by the emulator, focused on the emulation of the memory subsystem. The guest physical memory is composed of pages, so the emulator is managing corresponding pages in its own virtual memory. Indeed, the emulator executes as a regular process and can therefore allocate pages of memory. The emulator also needs to simulate the guest hardware MMU in software, this is the role of the software MMU tables. The first table translates guest virtual addresses to guest physical addresses while the second table translates guest physical addresses to host virtual addresses.

Still, on Figure 2.5, we can also see the guest instruction and its translation into the code cache. The guest `ldr` instruction can be found in one of the physical page of the architected state. Before being executed, it is translated into a snippet of x64 instructions to carry the call to the helper function, a C function available in the code of the emulator. The helper function navigates through the tables of the software MMU in order to translate the address at which the `ldr` instruction performs an access. If the guest MMU is enabled, the address is a guest

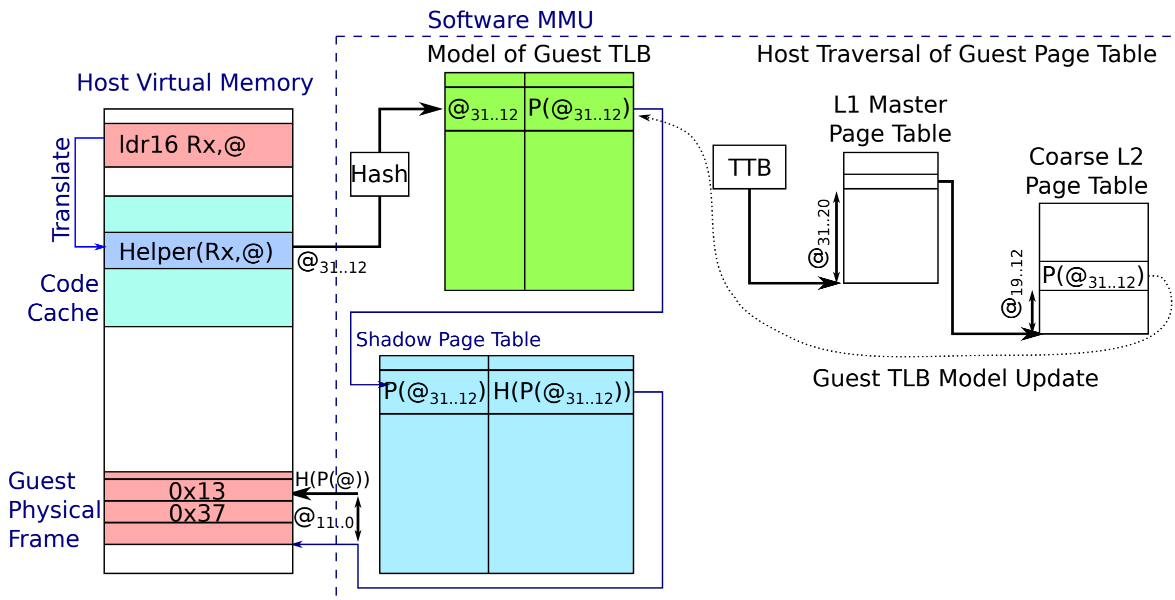


Figure 2.5: Host view of the architected state of the guest

virtual address, first translated to a guest physical address, and finally translated to a host virtual address.

The software MMU is also responsible for emulating memory-related traps that may occur during the address translation or the actual memory access. The address translation may fail due to a missing mapping or an invalid access right. The translated address may also turn out to be an invalid address in the guest memory map. Overall, this means that the emulator must emulate the guest processor behavior for traps, emulating the jump through the vector of trap handlers.

While this emulation approach works, its performance can be improved. Helper calls and the software page walks induce an important overhead. Because the code of the software MMU is fairly large, the impact on the instruction caches is non negligible. Furthermore, the translation always happens, always walking the page tables, inducing memory accesses that may trash the data caches and therefore negatively impact the performance. These overheads have suggested an optimization in software, inspired by the hardware Translation Look-aside Buffer (TLB). It is organized as an inlined fast path and the slow path being the call to the helper function. The inlined fast path is just a few instructions and a few memory accesses, looking up in a small table that caches the latest translations, exactly like a hardware TLB would do. Unsurprisingly, this table is called the software TLB.

The code of the fast path can be seen in Figure 2.6. The first two instructions are not strictly part of the Virtual TLB code, but are in charge of retrieving the guest register value and add the offset of the target load instruction (4 here). The ten next instructions are the real Virtual TLB code. First we compute the index at which the guest address should reside within the Virtual TLB. This is a simple hash where we take the N first bits of the VPN, N being the base 2 logarithm of the size of the Virtual TLB. Obviously, just as with a hardware TLB, it is possible that the hash will collide, and thus that the address at the index will not be the right one. To check this we also compute a tag, which is the rest of the guest VPN. Those steps are covered between line 2 and line 7 of generated host code. Then, at line 8 we compare the computed tag with the one from the Virtual TLB entry. If it matches, the jump at line 10 is

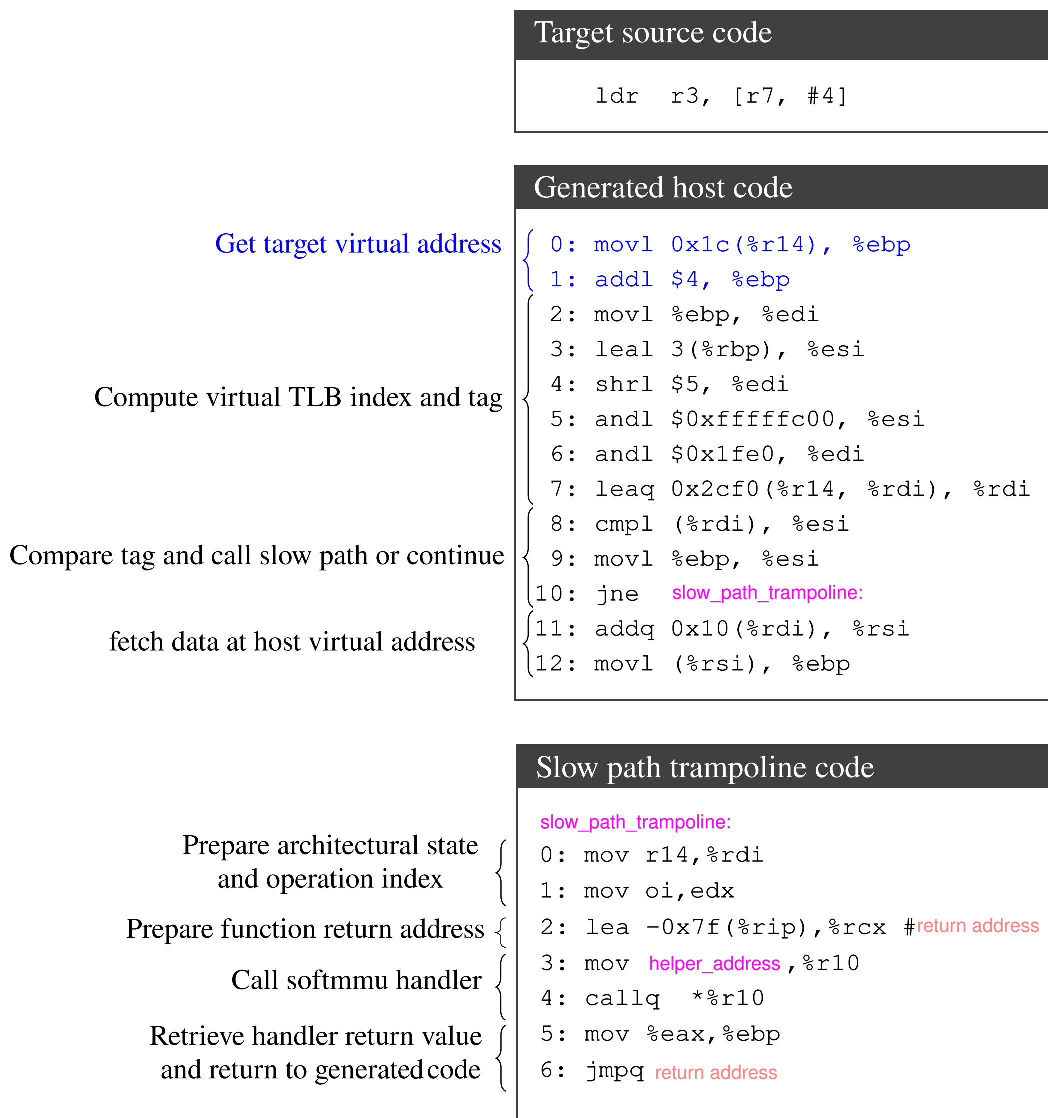


Figure 2.6: Qemu fast path code

not taken. Execution continue to line 11 where we retrieve the host address corresponding to this entry. And we execute the access at line 12. However, if the tags did not match, we now have to call the slow path. This is done by first calling a short trampoline code, always placed at the end of every translation block, which prepare the Soft MMU handler arguments, and then actually call it (line 0 to 4 in trampoline code). Once the handler returns, at line 5, the trampoline retrieves its return value and place it in the right register (to correspond to generated host code). Then, we jump at the next instruction in the code cache. So, even in the best case where we have a Virtual TLB hit, we will execute 11 instructions purely for the Virtual TLB, which includes three memory accesses and one jump. In the end, the software TLB improves performance, but it is not ideal. The code to access it contains much more than the original single guest instruction, either a `ldr` or a `str` instruction. Furthermore, whenever the fast path fails, the slow path is still a full call to the helper function, with a full address translation done in software.

To summarize, we have seen that emulating target memory accesses is actually fairly

complex. A whole architected state has to be maintained so as to track the location of the target physical frame and the exact target MMU state. The code to access target memory then potentially includes a full walk of both the target page table and the simulator shadow page table. An acceleration do exist to avoid constantly doing this. However, this optimization is still far from the original code in term of size and performance.

2.3 Conclusion

Simulating an instruction set architecture is a task that dates back to the infancy of computer design. In this chapter, we have seen that multiple instruction set simulation techniques exist, with different speed of execution versus accuracy of non-functional properties (such as time or power) trade-offs. Also, among these techniques, not all have the ability to perform cross-ISA full system virtualization. For that goal, dynamic binary translation, thanks to its property, is the solution we select. It offers support for unmodified target programs, including OSes and binary only (legacy) code, as well as a decent speed. Its accuracy in term of estimation of non-functional properties can be made reasonable [GFP09]. Yet, it also faces its own performance problem. Amongst those, one has particularly caught our attention, target memory accesses simulation. We then ask ourselves the following questions:

1. What schemes may be devised so as to accelerate target memory accesses simulation in dynamic binary translation?

2. What is the most efficient method we can think of to implement those schemes?

3. Are the designs we choose to accelerate target memory accesses simulation implementation specific, *i.e.* tied to the underlying OS or framework we use, or can they be reused atop any of those?

Chapter 3

State of the art

IN our problem statement, we introduced dynamic binary translation as an efficient mean to perform efficient instruction set simulation. This technique spans far more than that, and in this chapter we briefly present an overview of other applications of DBT that demonstrates its interest. Indeed DBT, which relies on compiling dynamically a target binary machine code into a different binary machine code which will be executed on the host machine, is at the core of multiple other technologies. Those mainly are high level languages virtual machines, code instrumentation, host specific dynamic optimization, low energy consumption instruction level parallelism optimizations and fast CPU simulation.

With this landscape settled, we present the various techniques that have already been used to optimize the speed and efficiency of dynamic binary translation. And finally, as this is the main scope of this work, we present techniques that were devised specifically to optimize target memory accesses simulation.

3.1 Dynamic Binary Translation

Dynamic binary translation, which is also sometimes references as Just In Time compilation (or JIT) in its general form is a technique used to translate, at runtime, a binary code to another one. It may be used either to translate machine code from different ISAs, as is the case in simulators, or high level languages bytecode to host machine code as is often the case in high level languages.

Virtual Machines

One of the very common use of dynamic binary translation is for high level languages virtual machines. The first use of DBT in general was actually the virtual machine made for the smalltalk language[DS84]. In this work, the authors try to obtain a fast implementation of the smalltalk-80 system. For this, they already use a bytecode as an intermediary representation of their program. However, instead of being interpreted, this bytecode is dynamically translated to host binary code. This in turn allows for the reuse of the generated code, and leads to important speedups compared to pure interpretation. Their inspiration came from [Rau78] which already exploited the fact of compiling a high level language to an intermediary one. Then this language is dynamically, or equivalently “just in time”, translated to host code. A cache, smaller than the overall program, is then set up to contain the most often used subset of the translated code.

One of nowadays most used programming language, Java, also uses DBT to speedup its execution. And in particular, in its server version [PVC01] which is a performance critical component. A difference with smalltalk is that, especially in the server version, many aggressive optimizations were added, such as code inlining or complex graph coloring based register allocation. This, in many ways, brought those languages closer in term of execution speed to compiled languages.

Legacy Code execution

Another popular use of dynamic binary translation has been the support for legacy code execution, or interoperability in general. The idea there is to execute a binary application, or potentially even an OS, made for an ISA that is not compatible with the current host. This can be used whether for compatibility with a legacy architecture, examples of those are Transmeta's Crusoe [DGB+03] which is able to run any x86 binary unmodified on a custom VLIW host and DAISY [EA97] which can do the same for PowerPC and potentially other architectures. These two solutions are able to simulate target OS code, or even bios. Other works, such as [BDE+03], aim at providing the capacity to run x86 unmodified appliactions on IA64 CPUs, and Aries [ZT00] which does the same for PA-RISC ISA to IA64, only focus on application level compatibility. We cannot end this paragraph without mentioning Rosetta, from Transitive Technology (a spin-off of University of Manchester [RS97]), that ran applications compiled for PowerPC on x86 from 2006 to 2009 on Macintoshes.

Code Instrumentation and tests

Popular code instrumentation and test tools such as valgrind [NS07] or DynamoRIO[BZA12] also use dynamic binary translation to speed up their analysis. Usually the principle of those is to add, at some specific checkpoints in applications (memory accesses, memory allocations, memory deallocations), pieces of instrumentation code which allow to ensure the application behaves as expected. Valgrind can also be used to estimate program performance across various metrics such as cache utilisation and hit ratio or branch predictor misses. DynamoRio offers this possibilities plus other functionalities such as backward compatibility analysis.

Recompilation, or binary optimization

One of the historical usages of DBT has been to improve applications performance. As an example DynamoRIO, before being used as an instrumentation tool was used as an optimizer [Gar03] for x86-IA32 code. DynamoRIO itself was based on dynamo [BDB11] which provided transparent optimization of HP-PA binaries on HP systems. The main hypothesis those optimizers based themselves on is that static analysis of compiler is no longer enough due to the late binding approach of newer executable[BDB11] (object oriented programming, dynamically loaded libraries). To improve performance, the idea is then to make optimizations that catch performance problems which could not be seen by the compiler. Example of those mainly being indirect branches, virtual function calls and redundant jumps and calls due to those.

Other works such as [KCB07] propose to cooperate more directly with the static compiler, with the static compiler preparing optimized code templates to be instantiated at runtime. Finally, works such as [ECC15] which is based on [CCL+14], continue on this trend of cooperation but with the goal of improving program interoperability and performance. This can be particularly useful given the jungle of SIMD instructions that are available on different

generations of processors. Basically, most of the program remains in standard C code, but performance critical sub parts which require host specific features are written in a RISC like intermediary language. At runtime, this intermediary language is translated to performance related host specific instructions.

Hardware optimization

Another optimization use of DBT is hardware co-designed DBT. This is a relatively new trend which aims at bringing DBT to native or superior performance in multiple scenarios. The principle of those solutions is to take care, in hardware, of various parts of the DBT. Those are typically used in an optimization context where we translate a RISC based ISA to VLIW based ISA so as to reach performance near to that of superscalar CPUs with a much lower energy footprint. An example of such works is [RRD17]. In this work, the authors are using a Mips target over a VLIW based host. What this work mainly aims at is to discover instruction level parallelism in the Mips target code, and schedule those instructions in parallel on the VLIW host. This solution obtains up to eight times speedup over a software implementation of the DBT while consuming eighteen times less energy. These kinds of approaches are also used in industry grade solution by NVIDIA[BBTV15], but with an ARM target.

Full System Virtualization

Finally, many tools used for full system virtualization are based on DBT. Amongst them, one that probably is the most used nowadays is QEMU [Bel05a]. It offers a full system simulation mode, with the capacities to simulate many types of devices. As of the time of this writing, its DBT engine is able to simulate 21 groups of ISA (with 32 and 64 bits variants), and it can simulate popular computing devices such as a PC board or a Raspberry Pi. The core of its technology relies on a front-middle-back end design. The frontend is mainly composed of target instructions decoders for each ISA, which decodes target instructions and translate them to the middle end intermediary representation. The middle end then makes an optimization pass on the IR and transmits it to the backend. The backend finally translates the IR to host binary code. This design allows to decouple frontend and backend completely. Thanks to this, the simulator is relatively easily retargetable both for new target and host architectures.

To give a more precise idea of this accomplishment, only Simics [MCE+02], which is also a full system simulator, seems to be able to handle a comparable quantity of targets. And Simics, at the difference of QEMU, is based on interpretation, with its interpreters being generated from a high level description instead of being hand coded.

While it is the most widespread, QEMU is not the first tool of this kind. We can cite Mimic[May87] as probably the first one to use DBT for simulation, which was made to simulate a full System/370 based computer. Embra [WR96], which itself is based on Shade [CK94] then simulated a full MIPS based board with multiple CPUs and their caches. Even today, other simulators based on DBT still exist. An example is CAPTIVE [SWF16], which also gained the ability to leverage HAV in a cross ISA context. We will detail CAPTIVE more precisely in section 3.3.

3.2 Accelerating Full System Dynamic Binary Translation

We now interest ourselves in the techniques which have traditionally been used to accelerate dynamic binary translation. Those are first the optimization of generated code, with mainly the introduction of trace based generation. This is often done using another thread so as to avoid stalling execution. Then the support for mutli threading simulation itself has been fairly sought after as it allows to potentially multiply performance by the number of target cores.

3.2.1 Generated code optimizations techniques

Many of the optimizations techniques here were devised in works concerning runtime optimizers based on DBT, but those can also be applied to full system simulation.

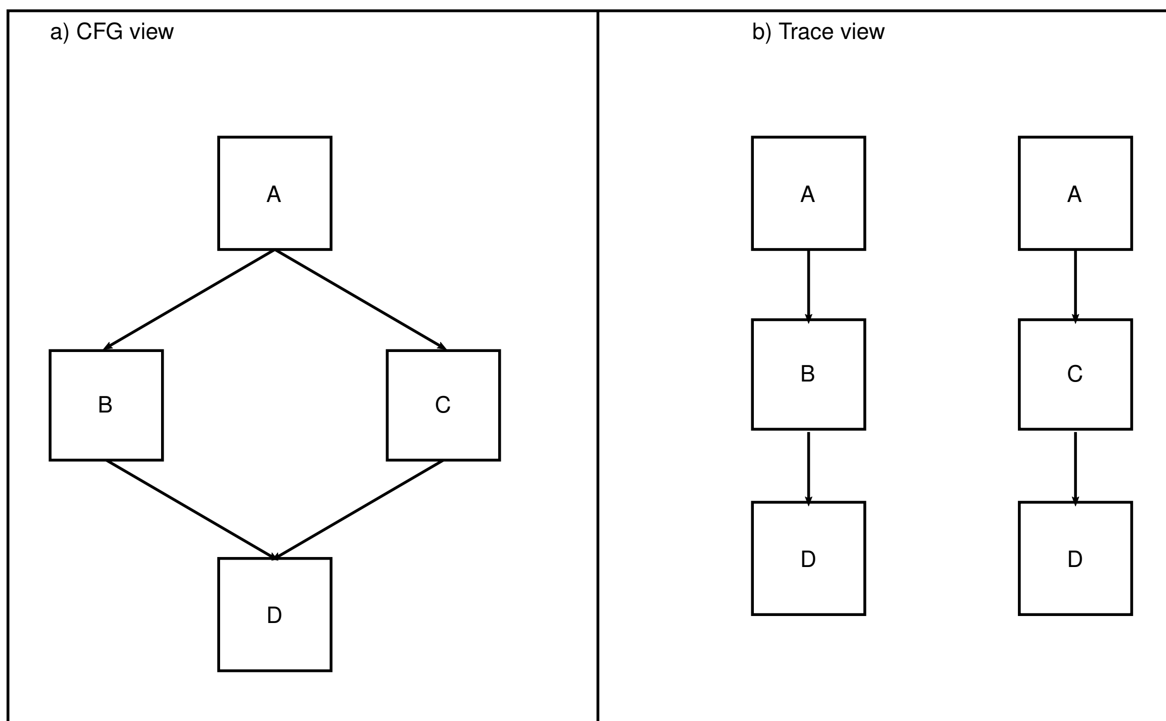


Figure 3.1: Example of a simple CFG and its equivalent trace view

Before delving into these optimizations, it is important to explain the two main representations of the code a DBT code generator, or more generally any compiler, may have. Figure 3.1 supports this explanation.

On the left, we have the standard form that is used in compiler, the CFG (Control Flow Graph). This representation partitions a program according to its branch instructions. Basically, each rectangle in the figure represents a continuous sequence of instructions that does not contain any branch but the last instruction, that is a branch instruction. These rectangles are termed "basic blocks" in compiler writers parlance. A basic block points to its successor block or successors blocks, which themselves point to their successors, and so on.

On the right, we have a trace based view of a program. In this case, there is still weak notion of block. However, the notion of successors disappears. A trace is a continuous flow of blocks, without any branching outside the trace. It is for example possible that a loop be

one trace, with the end branching back to the beginning of the trace. On the other hand, if a block has two successors, then only one can be part of the trace. A different trace will then exist for the other successors.

One such example is optimization work made by DynamoRIO[Gar03]. The base idea in this work, and most others, is to go from the usual compilation CFG-basic block paradigm to a trace based approach. In its basic approach, DynamoRIO still relies on a basic block vision of the target code. It decodes the target binary instruction per instruction and stop whenever a branch happens. The problem with this approach is that at the end of every basic block, control must be transferred back to the DBT engine, which implies a full context save, a hashtable lookup and then state restoration.

To enhance performances of the optimizers, the authors changed this process with the aim of reducing the number of control transfers. The first level of optimization is to link generated blocks which use direct jumps to each other directly. This already allows to avoid control transfers in simple cases. However, the problem remains for indirect branches (branches with a register that contains a pointer to the target as an argument). Those can not be solved at generation time, as such the previous optimization can not be used on them. Also, while direct jumps are faster than a control transfer, the jumps can still potentially be avoided in a trace, and other performance metrics such as cache locality might suffer compared to target code original layout.

As a solution to these problems, the authors changed the code generation so that basic blocks which usually execute in sequence may be squashed together in a contiguous block. This block being called a trace. Then, if one of the basic blocks contained in the trace ends in an indirect branch, an in cache check is made to check whether the target remains (or not) in the trace. If yes then no control transfer will be made and we can expect a performance enhancement. To make the traces even more relevant, procedure calls are also inlined in full.

Overall, according to the authors' analysis, the direct linking of basic blocks has an effect as dramatic as caching translated code (*e.g* going from interpretation to DBT). Traces and in cache indirect branch checking then add a significant performance boost.

Optimizations on the generated code itself were also applied, those mainly being redundant load removal, which as its name implies, removes load from memory to register when the value already is in a register. Architecture specific optimizations are also added. Those consist in choosing generated instruction according to the exact CPU architecture used by the host. Reusing the authors example, `inc` and `dec` are slower on a pentium 4 than `add 1` and `sub 1`. However, those are faster on a Pentium 3. As such, the optimizer will choose which instruction to use at runtime, depending on the host CPU. Finally, the authors implemented indirect branch dispatch, which consists of replacing the indirect branch lookup with an inlined fast path for common target. This fast path is a sequence of compare and conditional branch, with at the end a hashtable lookup if none of the inlined targets were correct.

An improvement to the trace generation method was later done in [GF06], in the context of a Java VM. In this work, no basic blocks are ever generated. Whenever the VM tries to generate code, it only generates a trace. For this particular work, the span of traces was chosen to be loops; *e.g* the beginning of a trace is the entry of the loop, and its end, the exit of same loop. As traces are generated they begin to form a trace tree, which is a group of traces sharing the same root.

According to the authors, in addition to the benefits which could be expected according to [Gar03], the main advantage of this method is that compilation is much faster. The authors confirm their code generation is made in linear time. Even more, they manage to produce code that is in a specific optimization format, that is SSA (static single assignment). SSA

being a special set of constraints on the code allowing to perform optimizations and register allocation much faster than in the general case.

3.2.2 Multi Threading support

Similarly to many other types of applications, simulators can profit from the fact that modern CPUs possess multiple cores. To do so, the simulators must be rethought to parallelize subparts of their execution. Two main paths have been explored, one is to parallelize code generation so as to make it parallel to its actual execution. The other is to parallelize the simulation of multi core targets, with each target core having one corresponding host thread.

Multi threaded code generation

A first approach to parallelism is to try to offload some of the complex tasks of DBT onto threads other than the main execution thread. One particularly complex operation is to generate high quality code.

Multiple works have tried to tackle this problem in multiple situations. Work from [BEvKK+11], is the first we found concerning full system Simulation. This work tries to heavily parallelize the code generation in the ArcSim[BFT10] cycle accurate simulator. This simulator is custom made for the ArCompact ISA and used LLVM[LA04] as its code generator.

The idea in this work is to decouple fully the code generation process from execution. Execution, at first, is made by interpreting target code. Then, as hot paths (or traces) are discovered, those are compiled to host binary code. To do so, code generation itself is heavily parallelized with multiple threads waiting to generate traces. Whenever a trace is discovered, it is placed in a working queue. This queue is sorted in order of priority, with the hottest and most recent path having the highest priority.

HQEMU [HHY+12] is another work that aims at using multi-threading in order to provide enhanced code generation in DBT, and in QEMU in particular. The authors' motivation can be explained through three observations. First, code generation in DBT must have a negligible impact on runtime to avoid slowdowns. Second, fast code generation techniques, such as the ones used currently in QEMU mono threaded code generation, lead to poor code optimizations. Efficient code generation techniques (*e.g.* those that provide high quality code) are time-consuming, and therefore not directly suited to DBT. Coming from those three observations, the authors designed a new solution which would allow to have both negligible impact on runtime and good quality generated code.

To that aim, the authors decided to split code generation in two phases, but kept code generation coupled with execution (*e.g.* code has to be generated before being executed). The first phase is a fast code generation phase which allows to begin code execution as fast as possible. This also ensures that no code is interpreted, even if outside hot paths, potentially enhancing performance for those as well. Then, while execution of this generated code is happening, a second code generation phase will optimize this code through more advanced optimization techniques. This optimized generated code can then replace the non optimized one in the DBT code cache. The core idea of this work is that both of those processes run in parallel on multiple threads, thus delivering larger potential speedups without the risks of a slowdown due to the code generator making the execution stall for too long. Thanks to this, the second code generator can provide various modern optimizations, such as trace based generation, instruction scheduling and code vectorization without having a negative impact

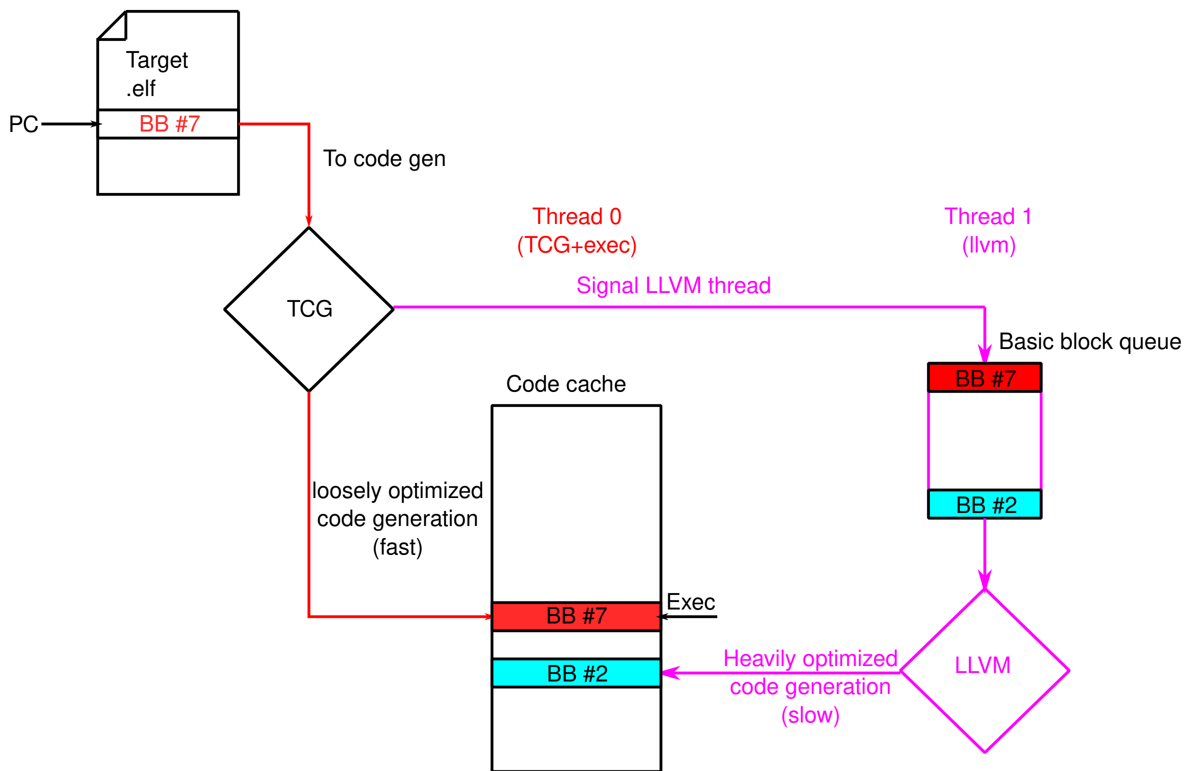


Figure 3.2: HQEMU threaded code generation

on runtime. In the end, this work provides moderate to large speedups on a varied set of benchmarks.

Another interesting part of this work is the performance analysis they made. There, they compared both their solution (with multi-threading), QEMU standard solution and a solution purely based on LLVM[LA04] for high code quality generation. What they show, compared to works from the section 3.2.1, is that a single threaded approach that only relies on generating high quality code can indeed cause slowdowns in quite a few situations. On the other hand, their solution which offloads quality code generation on another thread does not indeed cause any slowdown compared to a single threaded fast code generation only solution.

One important remark to be made is that only the code generation is multi-threaded, target CPU execution is still single threaded. In other words, no matter how many cores the target has, target code execution will happen only on one thread, with the target CPUs sharing this thread's execution time.

Multi threaded simulation

Another approach of parallelism, in particular in the case where the target has multiple cores, is to parallelise the simulation of target cores. A work which has aimed at doing so is [DCHC11]. The idea is conceptually fairly straightforward. For each target core, a matching simulation thread is created. These simulation threads will then be taking care of both code generation and execution on a core per core basis. The resulting execution, compared to a single threaded one is outlined by Figure 3.3. On the left part of the figure, we can see the standard single thread simulation. On one same thread, each CPU is executed alternately.

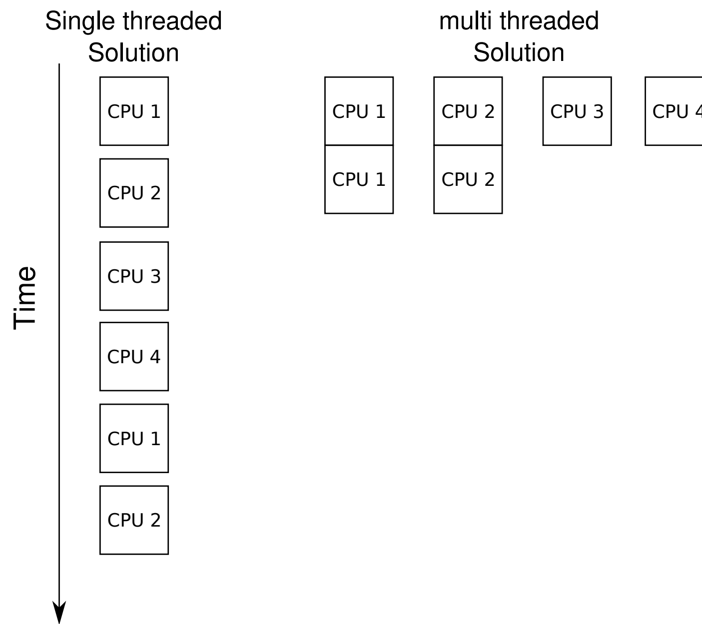


Figure 3.3: Example of a single threaded simulation execution versus a multi-threaded one

While one executes, the others wait, this means that, potentially, performance relative to the native target may be divided by NP, where NP is the number of target cores. On the right side, we now have the model where each simulated core executes in its own host thread. The result is that all of those may then run concurrently. As such, for the given example, the multi-threaded execution ends in two units of time (CPU 1 and CPU 2 need two units of time for their respective workload) whereas the single threaded one ends in 6 units of time.

Without much surprise, the real performance gain is fairly high for multi-threaded guest programs. With the author's benchmark, the average speedup is of 3.8 compared to baseline single threaded QEMU for a quad core target. All the gain is due to maximizing simulation throughput thanks to simulating every CPU at the same time.

One problem with this work however, as exposed in [RSR16], is that it did not yet support atomic instructions in a target agnostic way. Indeed, atomic instructions are not the same on every architecture. In particular, two different models exist. One is CAS (compare and swap) and the other code is LL/SC (load-link, store conditional). Emulating LL/SC with CAS, such as is the case when emulating ARM targets on Intel hosts, is non-trivial and require helper functions. Helper do have an overhead, yet this solution does outperform upstream QEMU in multi core scenarios.

3.3 Accelerating Memory Accesses Simulation

Finally, we present here works which have been made specifically to solve the problems of accelerating memory accesses simulation, mainly in DBT. Just the software translation of target addresses to host addresses have indeed been known to take 40% of simulation runtime on average. An explanation of why memory accesses are slow to emulate can be seen in section 2.2.3. Furthermore, not only are each memory accesses slow to emulate, but according to [JRHC95] about one instruction over two on a RISC cpu is a memory access. Having this observation in mind, it becomes clear that accelerating the emulation of memory

accesses is of utmost importance.

ESPT/HSPT

Two first works, ESPT[CWH+14] and HSPT[WLW+15], have tried to solve the problem of memory accesses simulation. They have mainly concentrated on embedding shadow page tables in the simulator process. The method is fairly similar between the two, so we will concentrate on the most recent one, HSPT, which relies on Unix `mmap`. Those two methods are using QEMU as their simulator, and modify it to fit their need. The main idea of the authors there is to have the fast path described in section 2.2.3 be reduced to one host memory access by embedding the target address space into the host one. This is done by reserving a part of the simulator address space and use a `mmap` syscall to map it on the same address space as the simulator (QEMU here) view of the guest physical frames. This mapping can then be mapped lazily, as the target tries to access it.

The base process followed to manage this mapping can be seen in Figure 3.4. At the

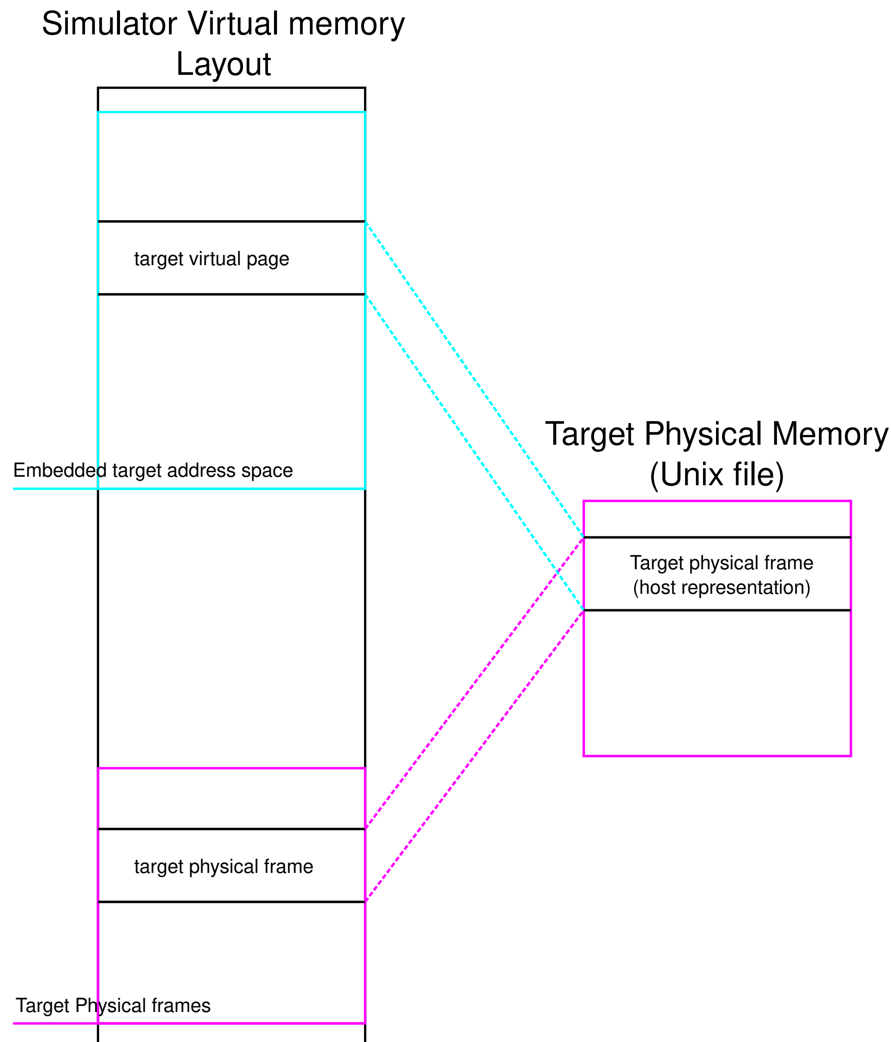


Figure 3.4: Memory layout of the simulator in HSPT

left, we see the simulator memory layout. There, we have both the simulator view of target

physical frames and the embedded target memory mapping. The physical frames are at the position QEMU placed them, but the embedded target memory mapping was placed by the authors, through the `mmap` syscall. On the right we see the file representing the target physical memory.

At first the embedded target memory mapping is mapped with the `PROT_NONE` flag, which means that any access will produce a page fault. Then, when the target tries to make a memory access it is redirected, through an offset, to the embedded mapping. During the first access, this will trigger a host fault, thanks to the `PROT_NONE` flag. This fault will then be forwarded by Linux to the simulator `SIGSEGV` handler. In this handler, a walk of the target MMU will be made, the corresponding target physical frame host address will be found, and the mapping will get updated through `mmap` to make it point to the right portion of the file representing target physical memory.

The embedded mapping then has to remain synchronized with the target mapping. To that aim, the chosen approach of this work is to track TLB flushes. Indeed, just as on a real CPU, the target CPU has to flush its TLB after any change made to its page table. To do so, the authors directly track the triggering of target instruction concerning those flushes, as well as changes to the registers holding pointers to the page table. With this, the part of the mapping which are to be updated can then be made unreadable and unwritable, thus triggering a fault again whenever an access happen. Then during this fault, the entry of the mapping will be filled with the correct mapping using the method we described in the previous paragraph.

This solution, which goes from a fairly simple idea, had to be slightly modified to remain efficient. The authors observed that flushing the whole mapping at every context switch was an important loss of performance, due to the fact that internal faults used to refill the mapping are fairly costly. To avoid this performance loss, the authors choose to retain one mapping per process. Thanks to this, when a context switch happens, the mapping is kept in memory, then when we context switch back to that process, we only have to use its offset in the fast path to access it, and internal faults number should be limited. The main limitation of this approach is that it relies on the presence of a Context Identifier in the target CPU (CID) which is a unique identifier for a given guest process. The second is that it can potentially lead to the use of a very important part of the simulator address space. For the second limitation, an optimization made by the authors is to only use a fixed amount of mappings, and when the number of process exceeds it, to reuse one (usually the least recently used one).

The second limitation, which further worsen the previous one, is that the approach was not thought with extending to 64 bits guests in mind. Nowadays Linux on Aarch64 is usually configured to use 42-bit long virtual addresses, or 2^{42} bytes of address space. This means that clobbering more than four flat view of the target address space would be near impossible with the usual process layout on modern operating systems. As such, the approach of limiting the amount of saved mapping would most of the time result in sharing one address space for all target processes. Actually, with such an address space size, and with it being bound to get larger, even maintaining a flat view of the target address does not seem practical.

Finally, a few differences with their previous work, ESPT[CWH+14] do exist. In the first work a Linux Kernel Module was used. However, it was only used as an augmented `mmap`. Basically, we can say that the module was never really used as a potential performance improvement in and out of itself. Faults were still communicated to the simulator through standard `SIGSEGV` handler, with all the technical and performance difficulties implied. A confirmation of this observation is that with the module, or with an `mmap`, the authors obtain roughly the same performances.

Another interesting analyze to be made is that the fast path of the first work is actually

slower (it still contains a branch instruction). Yet, performance is still the same. This is a sign that the fast path performance in itself is not the only dominating performance factor. Quite probably, the cost of the page faults has an at least as large impact on performances.

This page fault cost should also be much larger nowadays, whether with `mmap` or a module. Indeed, the totally unrelated topic of OS vulnerability with the recent meltdown [LSG+18] attack has lead to a change in the way OSes handle system calls and faults. OSes running on top of Intel CPUs now need to have two separate mapping for kernel space and user space [GLS+17]. What this means for us is that syscalls (`mmap` or custom for ESPT) now require a context switch in addition to a mode switch whenever they are made. This is bound to have a negative effect on target code which causes many faults, such as IO bound code, or with many protection rights.

HAV based cross ISA full system Virtualisation

Another method, based on an ad-hoc simulator named CAPTIVE, as described in [SWF16] relies on using hardware assisted virtualisation to provide a platform for cross ISA simulation. The base idea is to isolate target code execution onto a virtual CPU where it can have its own address space, and thus avoid the usual runtime checks on every memory access. An overview of the resulting layout can be seen in Figure 3.5. As can be seen there, the overall

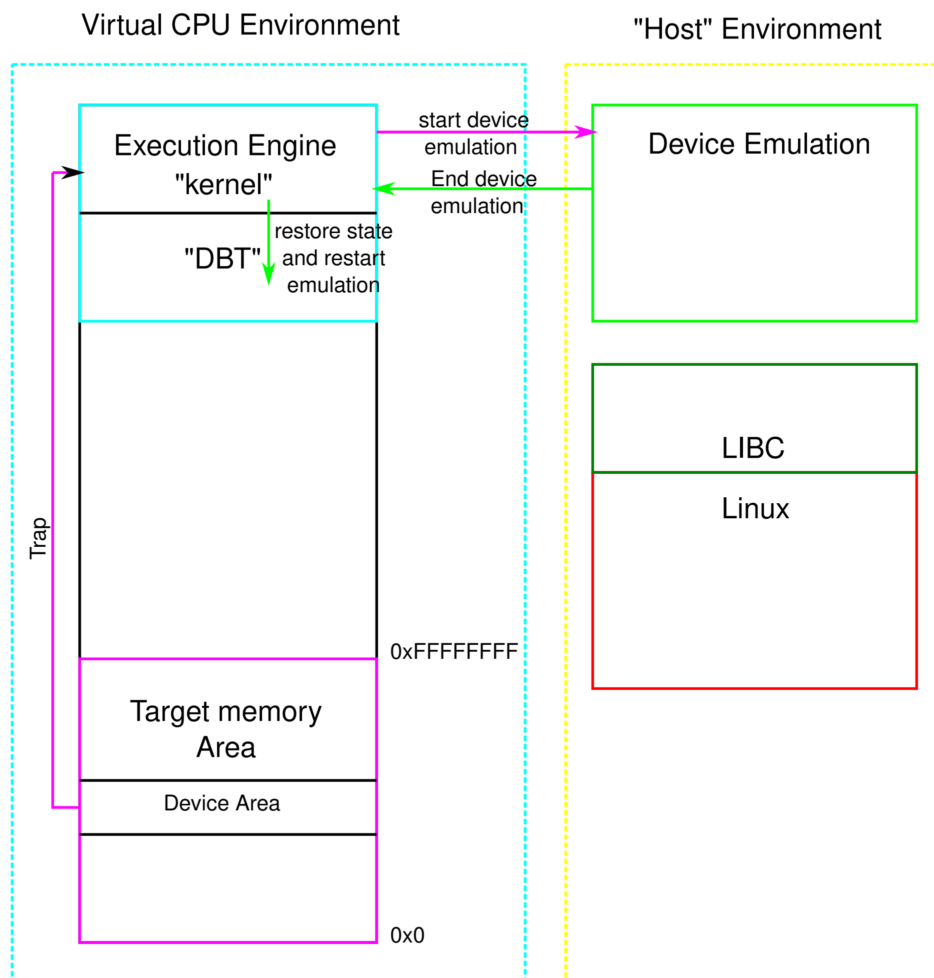


Figure 3.5: Memory layout of the CAPTIVE simulator

principle is as follows: On the left, we can see the virtual CPU current memory layout. From address 0x0 to 0xFFFFFFFF the address space is reserved for the target address space. Just as in HSTP[WLW+15], this allows to make memory accesses with potentially only one host instruction. But one limit that immediately comes to mind is the ability to scale to 64 bits targets. Indeed, in such cases, it seems highly complicated to map the target address space in such a way. In that case, the approach would quite likely require supplementary indirections in the code.

On the higher half of the memory layout lies the execution engine, which is an x64 kernel managing the simulation. Its rôle is to manage the overall simulation process. Basically, it contains the needed dynamic binary translator to translate code from the target ISA to the host ISA, plus all the DBT management code. That is, interruption simulation and communication with the host simulator code. On the right side of the figure, we can see the host simulator. Its main rôle is to perform device simulation. Usually, devices will be represented through the use of C code callbacks which emulate device behaviour while keeping a state automaton of the device.

Once again, the base principle is fairly simple, however its implementation is not. What seems like the main difficulty for the authors is to communicate with the out of HAV part of the simulator. Indeed, the authors choose to keep their device emulation thread on the host environment, and not with the execution on the virtual CPU environment. The problem is that one of the main performance costs in the case where HAV is used is to exit from the virtual CPU to the normal host process. According to work from [SFGP15] which implemented similar approach for native simulation, this can cost in average ten thousand cycles. The authors from [SWF16] confirmed the problem and devised another strategy to implement communication with their device thread on the host environment. First, every part of the address space which are device related are filled in the virtual CPU page table so as to trigger a trap in the execution engine (which acts as a kernel here). This is the "device area" part in Figure 3.5. Then, the simulator and the device emulation running on the host

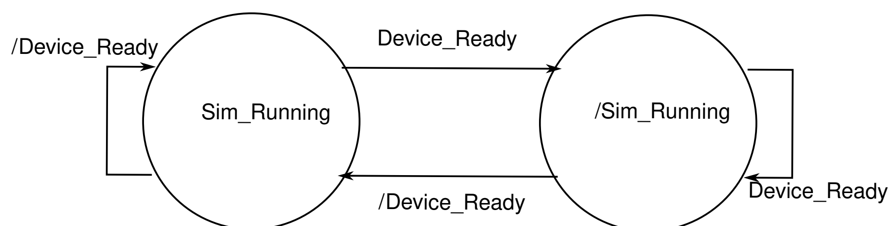


Figure 3.6: Handshake between simulator thread running on virtual CPU and device emulation thread on host.

part of the simulator will basically make a handshake, as described in figure Figure 3.6. The execution engine will interpret the trap as the signal "device_ready", and order the device emulation to start by entering a barrier. It will then sleep as long as it does not receive the inverse signal from the device emulator. At the same time, the simulator thread on the host part will begin to emulate the device access. Finally, it will signal the execution engine that the device emulation is done, and then enter a barrier itself to sleep until next device emulation is needed. The execution engine will then restore target state and resume generated code execution.

This principle seems fairly efficient for the monocore board simulated in this work. However, [SFGP15] has shown that in the case of multiprocessors, keeping the simulation synchronized leads to important simulation slowdowns. As such, going back to VM exits in

those case, and thus accept a larger performance hit, might be unavoidable.

Another problem of this approach is that it requires to basically write an x64 kernel. Kernels, no matter how small, are never easy to port to another architecture. This makes the solution heavily tied to Intel hosts. Also, debugging the DBT process in such environments is potentially extremely complicated, which potentially makes the development of new targets and backends for a simulator a pain, while it should be feasible by non-HAV specialist computer scientists.

3.3.1 TLB based solutions

The two preceding works give a fairly effective solution to the problem of emulating memory accesses. However, the main disadvantage of those two is that they rely on an important level of modifications to the way the simulator works. Thus, other works have instead chosen to only enhance the existing memory translation acceleration mechanism present in the translator. For QEMU, this is the Virtual TLB. The first of those work is [XTMA15]. This work led the first analysis we found about the memory emulation problem. According to the authors experiments, memory emulation usually takes between 30 and 40% of the whole simulation runtime. More precisely, 16,2% of the time is spent in the Virtual TLB lookup, and 15.9% is spent on page walks to refill the Virtual TLB. To reduce this overhead the authors proposed multiple solutions. Amongst them were out of line fast path, a simple varying size TLB and finally the use of a victim TLB. The last one in particular is interesting as it is implemented nowadays in mainline QEMU. This approach indeed has the main benefit of being straightforward to implement while already bringing around 18% of performance improvement.

Another interesting analysis of that work has been on the use of SIMD instruction to speedup lookups. Indeed, while this approach could intuitively seem like an important vector of performance gain, practice tends to show a slowdown. This also brings down the idea of increasing the victim TLB "associativity". The term is quoted, because as opposed to hardware with which all entries are compared at once, in software half the entries of an associative array have to be traversed in average to find the item searched for[1]. So, with SIMD instructions, one could hope to divide the number of steps substantially.

Finally, they proposed dynamically resized Virtual TLB which seemed promising. As a matter of fact [HHC+16] implemented dynamically sized Virtual TLB in a more extensive way. Their main idea is as follows. First the execution of target programs are cut into sessions, those sessions being defined as the time between two Virtual TLB flushes. Then, at the end of each session, the occupation of the Virtual TLB is computed. If this occupation is above a certain threshold, then it will be enlarged for the next session. If it is below a minimal threshold, it will instead have its size reduced. Finally, if it is between those two thresholds, then the size will remain the same.

The implementation is made as detailed in Figure 3.7. The virtual TLB is now an array with a set maximum size, let us call it `MAX_TLB_SIZE`. At any time, only a part of this array is now used. This subpart is delimited by the `CURRENT_TLB_SIZE` variable, which delimits the usable entries from currently unusable one. We then change slightly the Virtual TLB lookup routine to take into account the current Virtual TLB size, as a variable, instead of as a fixed constant. The Virtual TLB lookup routine can then make use of it for the hash, and determine which Virtual TLB entry corresponds to which address given the current size. While we

[1] In vanilla QEMU the victim TLB contains 8 entries.

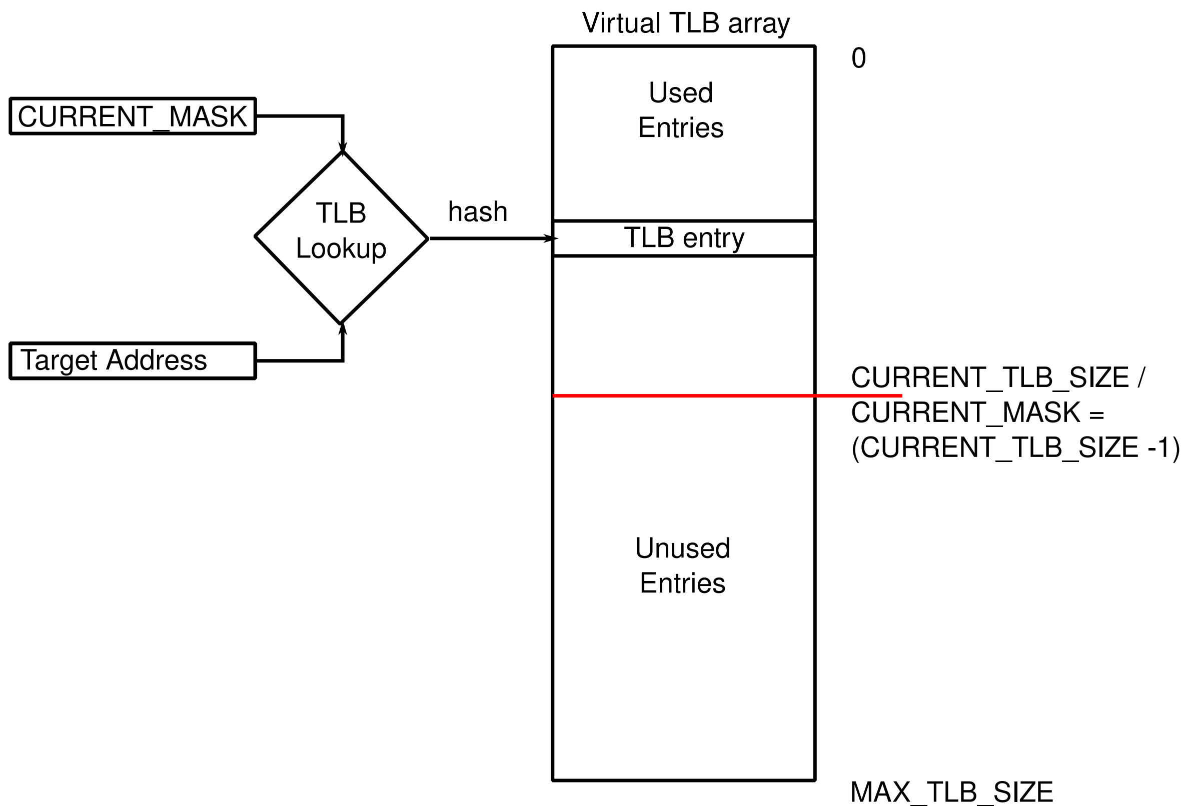


Figure 3.7: Detail of a resizable virtual TLB.

don't show the exact generated code, what it implies is essentially just a supplementary load instruction to retrieve the current Virtual TLB size.

The second part of the solution is to have a per page table size. Indeed, if the size information was global to all processes, the size would be adapted to the global use, and not to a particular process need. This would reduce the interest of the solution which is to provide a Virtual TLB size that minimize misses and flush cost (which is linear with respect to Virtual TLB size). To implement this, the authors simply use the target page table pointer. Indeed, as with each session end a flush occurs, even if the page table was modified, or if the identifier was not unique (reused pointer by the target OS), the simulation would still be correct. The only drawback would be that the next session would begin with an unadapted size.

A final optimization of the Virtual TLB made by the authors concerns the handling of large pages. Currently, QEMU does not really handle large pages in its Virtual TLB. Whenever a large page is encountered, only a sub part of it is mapped. Then, the whole address range of this large page is however registered in a second data structure, and when a flush that concern this range is occurring, the whole Virtual TLB is flushed. The problem with this approach is that flushes are fairly costly as they require zeroing out every entry of the Virtual TLB. This is even worst if the size is large. To avoid this, the authors implemented a partial flush strategy. The idea is to only flush pages according to their address range and no longer flush full mapping whenever a large page is caught in the flush.

The result of the combination of those two approach is a speedup of up to 1.89% on average (speedups depend on target ISA).

3.4 Conclusion and Remaining questions

In the end, we have seen that dynamic binary translation has many uses. Amongst them, full system simulation has been a heavy research subject as it is used either for virtualization or fast simulation. We have seen, in particular, multiple existing approaches that have been devised to accelerate simulation. Amongst them, the most researched ones are code generation optimization and multi-threading. Because of that, we consider both of them already relatively well looked after. Another approach, which consists in using hardware acceleration is also being studied by others, and while very interesting and promising, was not retained for this thesis which was thought in the context of retargetable simulation.

On the other hand, it has been analyzed that in the case of full system simulation 40% of simulation time, on average, is spent handling target address space management. Before this thesis, work has been made to develop multiple solutions to tackle that problem. In particular, ESPT and HSPT propose to embed target address space into the simulator address space. Concurrently to this thesis, CAPTIVE has also been designed to leverage HAV for cross ISA simulation in order to obtain similar results as HSPT and ESPT. We find that both approaches are promising in their own directions. In particular both reach the goal of having a fast path in simulation that is fairly short.

However, in the case of ESPT (and even more HSPT) the principle of a Linux kernel module assisting DBT was not truly exploited. The module does not provide anything more than a `mmap` like mechanism, which leads us to wonder why not use it to have more custom mechanisms to solve the specific performance issue of virtual to physical address translation, and overall target address space emulation, in DBT. Therefore, designing a module that is more active and specific to DBT will be our focus in chapter 5.

Regarding CAPTIVE, based on HAV, the main limitation we identify is that the simulator is split between the virtual CPU and a normal host process, which incurs complexity in term of communication between both realms. We also find that developing a kernel should, as much as possible, be avoided due to being tied to a specific host architecture. Designing a solution, based on HAV but using a normal Linux process that does not require communications between both worlds, will thus be our focus in chapter 4. Both of these solutions will also be shown to be implementable according to the same design (and potentially reusing most of the simulator code).

Chapter 4

Accelerating DBT using hardware assisted virtualisation for cross-ISA simulation

4.1 Introduction

IN this chapter, we give a first answer to the question "what schemes may be devised so as to accelerate target memory accesses simulation?".

What we want to obtain is a scheme that, in the context of full system cross ISA simulation, offers a performance overhead that is as low as possible on memory accesses simulation. To do so, we want to rely on host hardware to manage as much of the memory accesses simulation process as possible.

A solution to manage the memory access simulation, and more generally the simulation in hardware, exists: this is hardware assisted virtualization (HAV). It provides hardware acceleration for full system virtualization with almost no overhead. However, in its traditional form, it is only usable for virtualization of host and target sharing the same ISA. Also, as [CWH+14] argued, naïve use of HAV tends to create important communication cost between the hardware accelerated parts and the simulator.

Yet, the idea to use HAV still seems promising as it gives a full control over a new address space that is fully hardware accelerated (in particular address translation occurs as if it were occurring on the bare host CPU). This address space could then be used to accelerate the simulation of the target memory accesses. The main complexity is then to use the simulator in such a context, without having to pay the cost of communications between target code and the simulator and without turning the simulator into a kernel running on a bare bone virtual CPU.

Thankfully, this topic has been of interest to the operating system community, even though it was for a different goal: providing users transparent access to privileged CPU features. As a result, a team designed and implemented a tool called Dune [BBM+12], which allows running mostly unmodified Linux processes to run directly on top of a virtual CPU. In our context, this Linux process can be an existing simulator in which the memory accesses can be tweaked to use either the CPU or the virtual CPU memory management units. With not surprise, the simulator we choose is QEMU [Bel05b] since, as described in Chapter 3, it is widely available and widely used, is open source, is the one among all that we know of that provides the largest number of host and targets, and even though still in evolution, is quite

stable.

We will detail in this chapter how we use QEMU and Dune to provide a new solution in which the simulation of the target memory accesses is managed by the host hardware and for which the cost of communications related to HAV are avoided for all aspects of the simulation.

4.2 Background

Before delving into how we design and implement our solution, let us present the background techniques we used to achieve our goal. We will specifically detail how HAV functions, how the Dune framework uses it and the implications of using this framework for our work.

4.2.1 Hardware Assisted Virtualization

Let us first discuss the way HAV works and how to use it.

The base idea of HAV is to duplicate the view of the host CPU, creating two independent visions of the same CPU. One view, the root mode, remains the standard usual host view, where the simulator managing the simulation runs. The other view, guest mode, appear as a naked CPU, called virtual CPU and is the one which is used for virtualization itself.

On top of this naked view of the CPU, it is indeed possible to run an unmodified OS. The obtained layout is described in Figure 4.1 As can be seen on this figure, the guest OS really

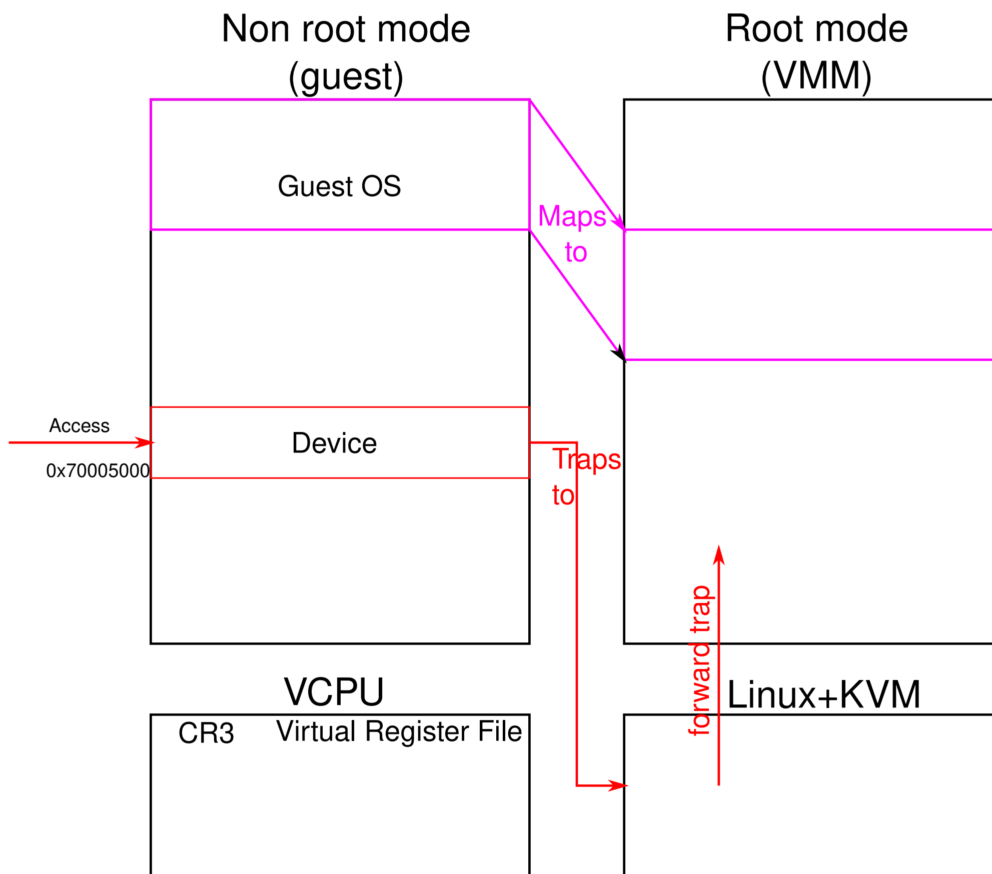


Figure 4.1: HAV memory layout and virtual machine exit

has access to hardware that is perfectly alike that of a real CPU. It sees a full address space, that it can modify as wished, it has direct access to all CPU architectural registers. Notably, the page table register (CR3) is also accessible, allowing the guest OS to manage its own page tables. This page table register points to a guest page table, allowing to make translation from guest virtual addresses to guest physical addresses. These guest physical addresses are then seen by the host OS (Linux on the Figure) and Virtual Machine manager (VMM, KVM+Qemu typically), as host virtual addresses.

Quite importantly, for most CPU types and specifically Intel ones that are typical simulation hosts, the guest page tables will be walked in hardware, and translations will be cached in translation lookaside buffers (TLB). The second part of the translation, from host virtual to host physical will also be chained directly, and done in hardware. This means that, for this simulation subpart, performance will be close to native. Such translation is called Second Level Address Translation (SLAT).

Also, the guest sees MMIO registers and device ports as usual. Typically, those devices will however not be the real ones but be emulated by the host. An example of a device access can also be seen in Figure 4.1. The guest, running in non root mode, attempt and access to the device at address 0x70005000. This access traps to the Linux-KVM host, and the trap is directly forwarded to the VMM. The VMM then executes the necessary emulation code for given device, and give control back to the guest. The guest is completely oblivious to the whole process. It is what we call a VM exit.

From the Host OS point of view, the virtualized guest is pretty much a user side process. It can schedule or unschedule it as usual. It handles its memory as it normally would, except that what it manipulates are in fact guest physical pages. And there are two page table registers that are chained, CR3 which gives guest virtual to guest physical translation and EPT (extended page table) which gives guest physical to host physical translation. The host allocates memory as normal as well. It can swap pages if there is not enough host memory and so on. The most important part being that all of this is fully transparent for the guest. Page faults on the other hand are slightly different. In case the page fault happens during the CR3 walk (guest protection fault for example) the fault will be transmitted to the guest, and not the host. Conversely, if the fault happens during the EPT walk (host swapped out a guest page) the fault will be transmitted to the host only.

Another difference is that context switches and traps to host, which are called VM exit, require to save the full architectural state of the virtualized process (everything in the VCPU in Figure 4.1, and in particular the virtual register file and CR3). Furthermore, traps, instead of going to the host OS, will go to specific entry points in the VMM, running in root mode. This, in turn, means that operations such as traps and context switches will in fact be much costlier than for a standard process. If the guest OS is unmodified, accesses to devices will be a large cause of such traps.

Indeed, devices are usually memory mapped (memory mapped input/output or MMIO), so that an access to a device traps on the host, which then handle the device access by executing its specific emulation code, pretty much in the same way as in DBT. If the device is complex, like a graphic card for example, and many accesses are made on those MMIO registers, then a large number of traps will happen, thus slowing down the simulation.

Virtualized OSes are however often paravirtualized, and are thus not completely oblivious to the host. In particular virtualized OSes can voluntarily call the simulator running in root mode when it is the most convenient for them. They could for example prepare a batch of operations to transmit to the simulator, and at the end only make the exit, which would process those. This is done through an instruction called `vm_call`. Multiple sources of VM

exit exist, the most standard one being device emulation. The idea of paravirtualization will then often be to reduce as much as possible the number of VM exits by paravirtualizing the devices which requires many trapping instructions. Example of a guest OS communicating with host through `vm_call` can be seen in Figure 4.2. In this figure, the guest OS is trying

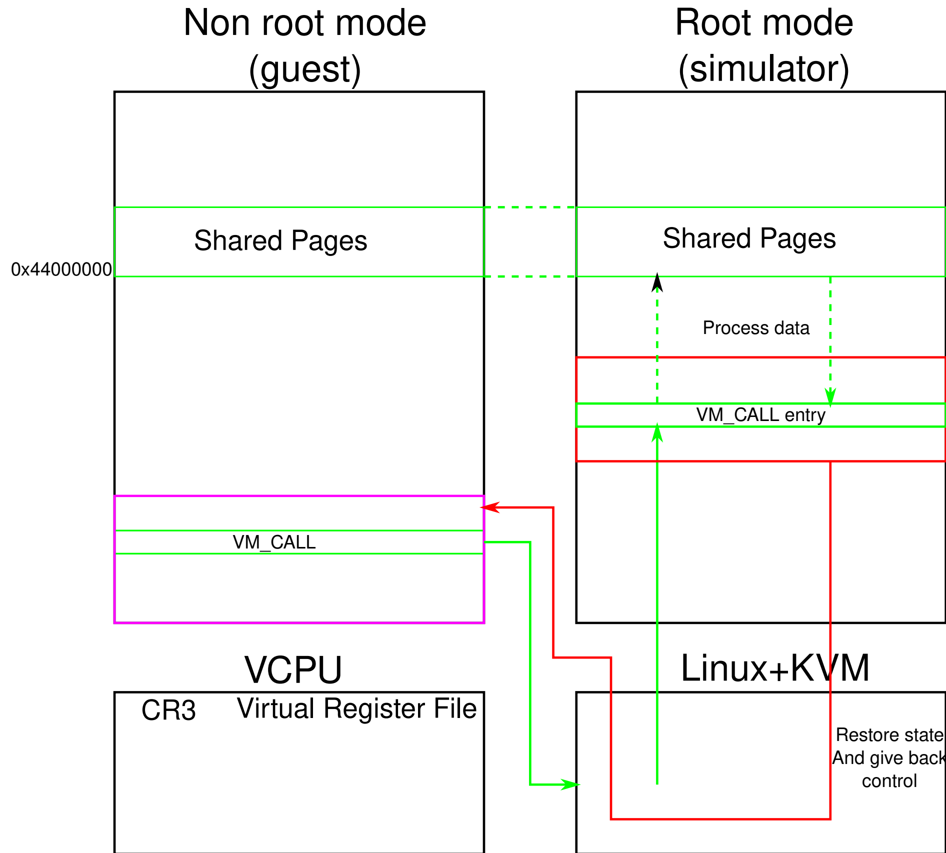


Figure 4.2: Principle of a `vm_call` based communication

to communicate to the host a batch of commands that it puts in a specific page at physical address `0x44000000`. First, it will write the necessary data at that address. Then, the guest calls the simulator using a `vm_call` instruction. The simulator then takes back control of the CPU. It then executes the commands as needed, and transfers control back to the guest. The main advantage of this method is that only a single round trip has been done. The other advantage is that, potentially, the commands could be executed on another thread, thus giving control back even faster.

4.2.2 Dune framework

Let us now explain how the Dune framework [BBM+12] builds upon HAV and its mechanisms to provide an augmented Linux process able to manipulate hardware directly.

The Dune framework[BBM+12] goal is to reuse HAV to execute an unmodified Linux process on top of a virtual CPU instead of an OS. The point being to allow user side processes to leverage hardware, and privileged instructions in particular directly. To do this, it builds up on paravirtualization support. It populates the non root mode CPU with the process address space, and its own library OS which provide the routines to make the virtual CPU

environment transparent to the process. Then the library OS catches the trap, and uses a `vm_call` to transfer it to the host. The way all the elements interact is described in Figure 4.3.

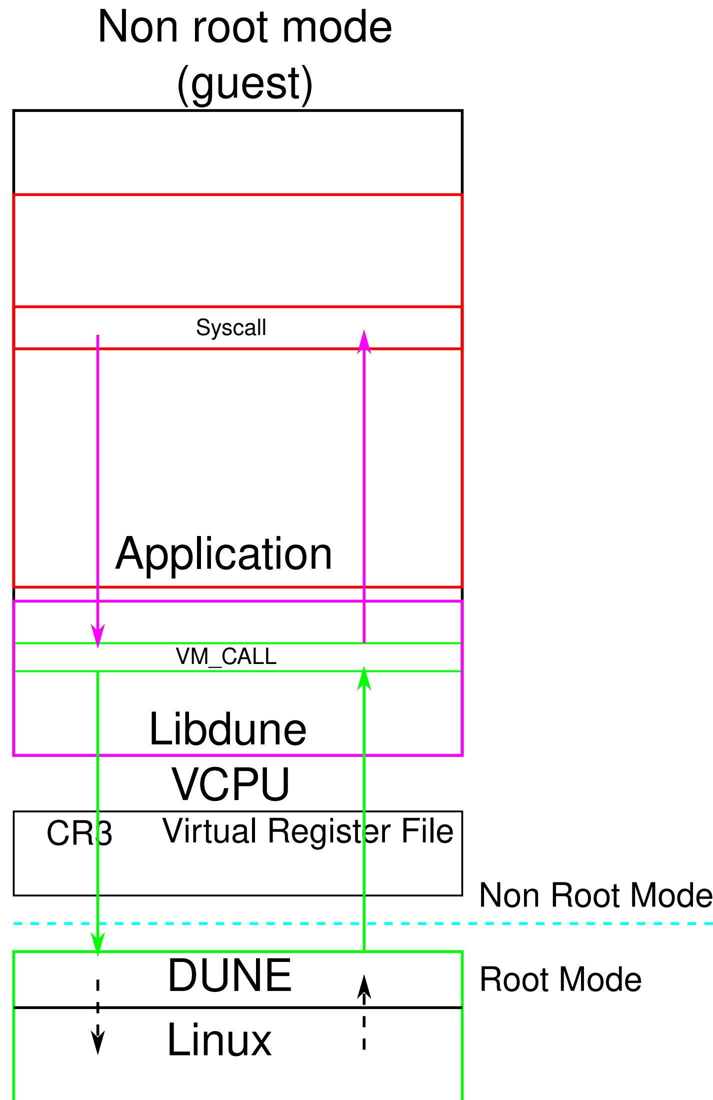


Figure 4.3: Process of a syscall in Dune

As we can see on the figure, the process is running on top of a thin software layer called libDune. This layer is basically a library OS taking the place of Linux for the process. This makes the process address space look pretty much just like a standard Linux process from its own perspective. All the internal program code runs just normally. However, syscalls no longer go directly to Linux. Instead, the Dune library OS catches them. Then it makes a `vm_call`, which is caught by the Dune module. This `vm_call` actually matches the syscall made by the program, and the Linux kernel module just transmits it as such to Linux standard syscall handling code. The result is then passed back to the library OS, which itself passes it back to the program. Applying this to our simulator, it can now execute on top of HAV.

Now the question is what exactly have we gained by doing so? In fact, while Dune allows processes to run unmodified on top of HAV, it is obviously not the main point for us, or even for the authors. Indeed, by default there, applications now run in privileged mode (on the

guest vCPU). As such, we have access to all the supervisor only registers, with in particular the page table pointer, and the interrupt vector table which allows to choose which handler to use for every type of traps and exceptions. This means that we can begin to set up our own fault handlers, and directly get called, without the costly intermediary of a kernel. As a reference, the authors of Dune estimate that a page fault handler in Dune is four times faster than a Linux SIGSEGV (the segmentation/page fault handler) to be called.

This also means we can modify our own page table and remap whichever part, however we wish. This in turn leads us to a path to reach our goal of accelerating memory accesses. If we can freely manipulate our address space, why not use this to map the simulated target address space on the host? This is what we have tried to achieve and will explain in following sections.

4.3 Design

Let us now study the design of our solution. As stated in the problem statement, our goal is to accelerate the simulation of memory accesses in full system DBT. In particular, we wish to do this by mapping the simulated target address space inside our simulator's address space. The goal of this approach is to reduce the *fast path* we described in section 2.2.3 to a single load or store instruction. A secondary objective is to leave QEMU mostly untouched so that our changes are minimal and easily incorporated in new versions of QEMU.

To move towards these objectives, the trick is to provide two memory translations within the QEMU process. The first one is the standard translation that QEMU code, as any process running on an operating system, expects. The second one is a shadow translation that the target code, generated by the DBT process, relies upon. The resulting memory layout can be seen Figure 4.4.

The first important fact to notice is that the standard QEMU translation remains untouched and fully functional. All of the QEMU offline code (*e.g.* code that is not in the generated code cache) can continue to use this standard path as usual without modifications. As an example here, the target page located at address `0x2000` is translated to the simulator virtual address `0x106000` by the standard Soft MMU Virtual TLB path. This page is a representation of a target physical frame.

At the same time, an embedded view of the target address space is included in the simulator address space. This area, which we here call "shadow mapping" is the one which will be accessed by the generated code. In our example this mapping is located at `0x10000000`, and is called Mapping Base Address (`MBA`). Now, the same target page as before, at address `0x2000` will be mapped at `MBA+0x2000`. Thanks to this, the generated code, instead of going through Soft MMU/Virtual TLB code path can new make a direct access to this mapping. We refer to this direct access to memory as an "inline" memory access. Now instead of executing the generated code presented in section 2.2.3, the access can be done using a simple `mov [MBA+0x2000], dest` instruction.

Now, the underlying of how this mapping works are presented in Figure 4.5. Looking at the left side of the Figure, what we see there is that, to make the shadow mapping work, we have to alias its pages to the same guest physical frame as the one used by the standard Soft MMU/Virtual TLB code path. The idea is that with this done, a fetch or a write to either `0x106000` or `MBA+0x2000` in our example will have the exact same effect. If a fetch is done, the same value will be returned, and if a write is done to either of these addresses, the underlying value will change at the same time for both, since they translate to the same physical address.

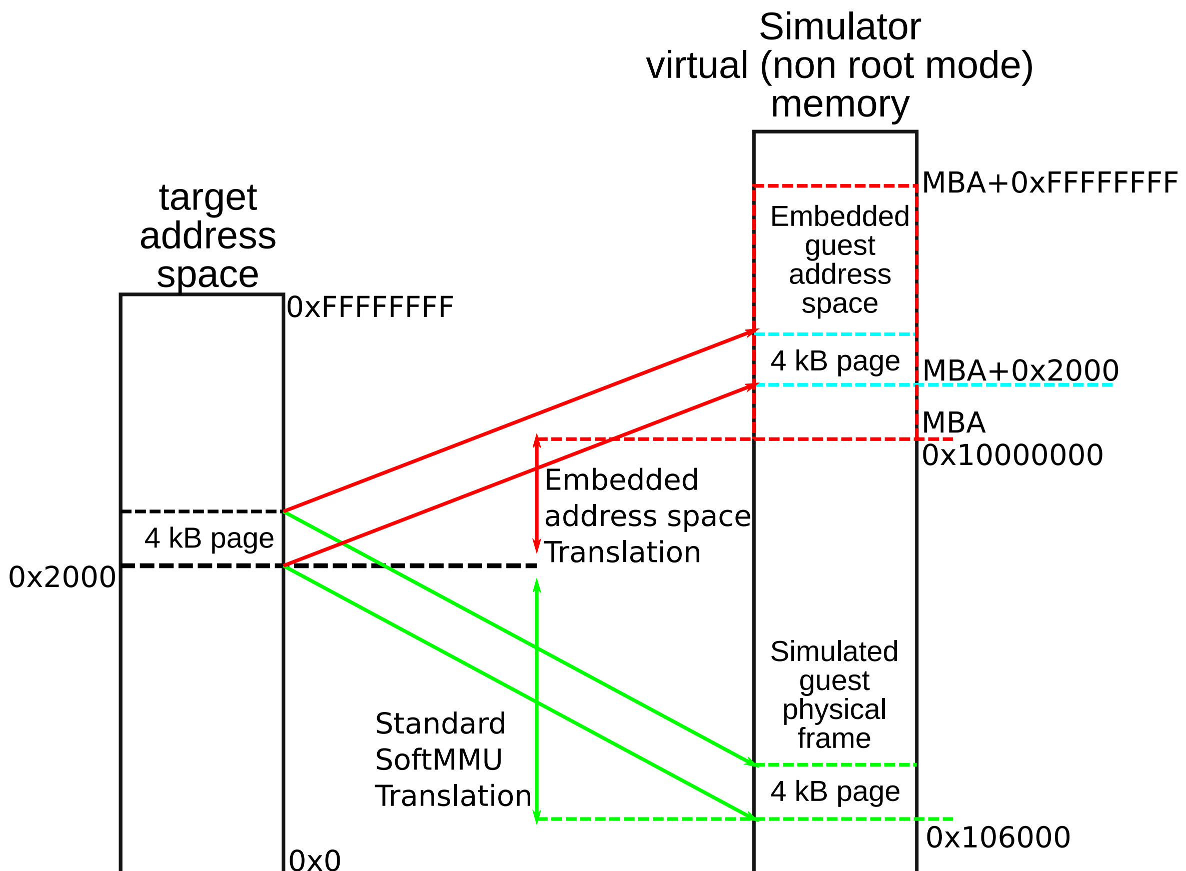


Figure 4.4: Memory Layout from the simulator point of view

One difference with the two mappings however concerns the access rights of each mapping. For the standard translation, the access rights are actually pretty much irrelevant, and will usually be read/write no matter what. The reason for this is that, in the case of this translation, a lookup of the Virtual TLB will be done before any access. This lookup will then proceed to an access right check to verify whether the target access is legal or not. However, in our case, the access is directly done inline, without any software check. This implies that our vision of the mapping must be made to have the same access rights as the target virtual page has. Thankfully, since we are running on top of Dune, we have access to the page table the Linux process running the simulator uses, and can thus manually mirror those. As a result, illegal accesses will trap, and the trap will be forwarded correctly to the target CPU running on the simulator by the simulator.

On the right side of Figure 4.5, the last level of memory translation is described. Indeed, the simulator is running on top of Dune, and is therefore, itself, a guest program. Because of that, a guest physical address need to be translated to a host physical address. While this causes a slight slowdown in virtual address translation process compared to a fully native program, it also has an interesting property. All the virtual addresses from the guest point of view are guest controlled. In other words, there is no guest page swapping that happens without the guest knowing. Conversely, any manipulation to host pages made by Linux will be invisible to the simulator. Even better, Dune takes the page faults right only for page table entries that were modified by the guest. As such, faults happening in non modified parts of our simulator will still be handled by Linux, with us being oblivious to them. This is

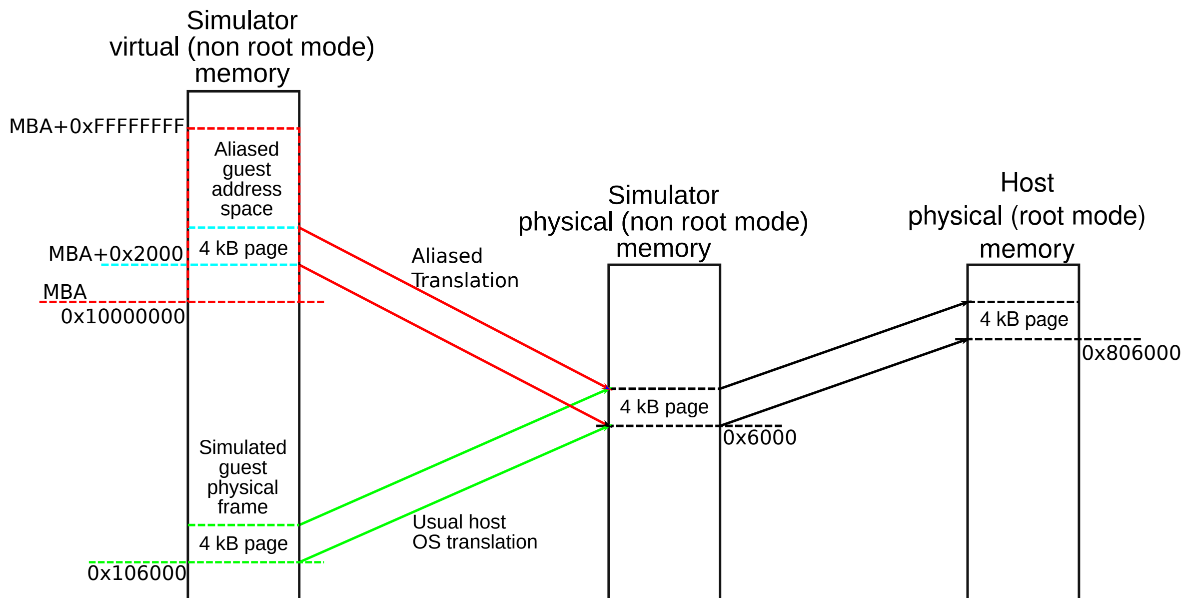


Figure 4.5: Memory layout from the guest physical memory point of view

particularly advantageous as it simplifies implementation a lot since we do not have to worry about changes that could happen to our or the standard virtual addresses translations (page swapping or page migration for example) or even to copy on write strategies developed by the process itself.

With this translation in place, target memory should now be directly accessible inline from the generated code. A first consequence is that it should bring a large speedup, since the size of the generated code and its complexity is drastically reduced. A second consequence is that, compared to QEMU Virtual TLB, our mapping basically has an infinite capacity from the target point of view, that is, every last target entry could potentially be mapped. This should in turn solve the problem of TLB misses for already seen translation altogether. Overall, at least in case of RAM accesses, we can now hope to obtain near native performances.

A problem might still however exist due to the communications between the host and guest. Just as [SWF16] we need to simulate devices. Most of those will actually use the Memory Mapped IO (MMIO) model *e.g.* device registers are directly mapped in memory. These devices will thus be accessed directly inline, but no mapping to a target physical frame will exist, as those are not RAM. At the end of the day, the access will trap. From this trap on we can leverage QEMU code to do the emulation of the device (more on that in Section 4.4). This means two things. The first is that compared to [SWF16] we do not have non-root to root mode communications, as such we can expect at least a simplification of the code here, and with multiple CPUs, maybe a performance gain as we could expect from [SFGP15]. The second is that however, we are getting a trap before every device access. Now, other embedded mapping methods such as [WLW+15] will have the exact same problem. They use a SIGSEGV handler to manage these cases. Yet, Adam Belay in [BBM+12] shows that a page fault in Dune is about four times faster than a SIGSEGV. As such, we can definitely expect an important reduction of the communication overhead compared to those solutions.

4.4 Implementation

Even though relatively simple conceptually, our solution needs to solve many intricate technical issues at the hardware/software interface. To ease the explanation, we consider throughout this section that we are working with an ARMv7 target and a x64 host. Other QEMU targets are however usable, as long as they use 32 bit addresses. Also, please note that whenever we talk of the "guest" page table, we refer to the page table used by the simulator process, which is a guest for the Linux host.

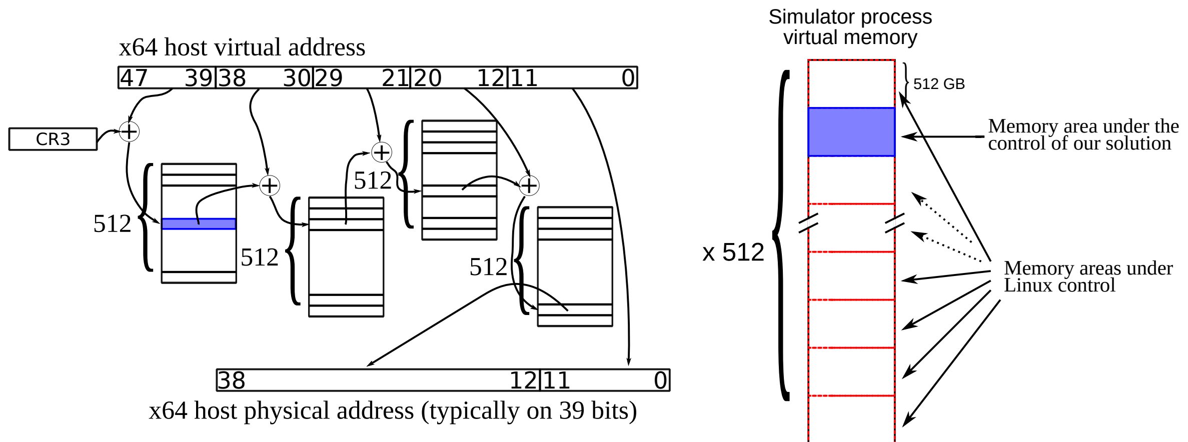


Figure 4.6: x64 (host) hierarchical page structure

What we mainly used for this implementation are the facilities provided by Dune, with in particular the guest page table pointer and interrupt vector. Using the Dune framework then enables us to only slightly modify QEMU in order to leverage the SLAT to create our own embedded target address space and install simulation specific page fault handlers.

The first step we have to take is to allocate the virtual memory we need to embed the target address space. To do so, we manipulate directly the guest page table. First we need to find a free location in the guest page table. Since we are simulating a 32-bit target, its address space will be no larger than 2^{32}. Now, if we look at the guest page table of an x64 Intel host, as illustrated Figure 4.6, it has currently (as of june 2018) still 4 levels (each being 9-bit long, plus 12 bits used as offset in the page), the virtual addresses are thus 48-bit wide. The `cr3` register points to the root of the page table, the page-map level-4 table, and thus covers 256 TiB (2^{48}) of memory. This is more than enough, but also not usable as the simulator needs to keep some address space for its own usage. Each page-map level-4 table entry covers 512 GiB, which is largely usable as there are 512 of them. It points to a page directory pointer table whose entries (PDPTE) point to the page directory, in which each entry covers in turn 1 GiB of virtual address space. As 1GB is too small for our use, we choose to reserve a full PDPTE entry for the target address space.

One important note to be made is that only page table entries we modify will be manged by Dune and our code. This is what we can see on the right side of the Figure. All the page table entries we did not modify remain in the control of Linux. Due to this, any fault happening in this zone will not be forwarded to us. Conversely, every entry we modified will be managed by us. If a fault happens that concerns one of those entries, then Linux will be oblivious to it. Our handlers will automatically be called, and we will be free to make any modifications we wish to it.

Now that we have chosen this solution, we have to generate the right code for memory

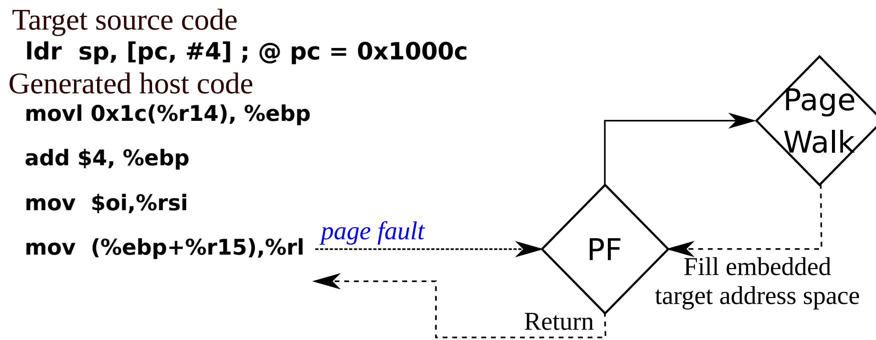


Figure 4.7: QEMU target memory access with our solution

accesses so that it uses this embedded target address space. This code can be seen on the left-hand side of Figure 4.7. The first two lines are standard QEMU code and are used to simulate the `pc,#4` part of the target instruction *e.g.* retrieve the value of pc and add 4 to it. The next line, `mov $oi,%rsi` corresponds to passing an argument to our fault handler, in case it needs to be called. The last line does the actual access, the fetch of a data here. The address of the fetch is computed using an offset, contained in `r15`. This offset is the `MBA` that we have introduced and explained how to use in section 4.3.

As can be seen on right side of the figure, page faults will also sometimes happen. Indeed, we fill our mapping in a lazy way. In other words, at first our mapping is empty, and whenever a first access happens, a page fault is triggered. At this time, the mapping will be aliased to the same guest physical address as the corresponding simulated physical frame in the simulator. Another potential cause of page fault would also be that either there is no mapping, because none can be made, or that protection on the page is violated by the current access (write to a read only area for example). This can be due to two causes, either the accessed memory is an MMIO register, or a target page fault is happening.

If the fault is caused because we did not yet fill the mapping, it is an internal fault, otherwise, it is a real target page fault.

Now the challenge comes from managing those faults correctly, which has to be done in the simulator page fault handler. Algorithm 2 gives an overview of the way a page fault is handled. Note that the available data when entering the handler are the following:

- io, which contains the mode in which the access was done (user/supervisor) and various state info for the memory access,
- pf, which reflects the cause of the page fault. More precisely, it contains:
  - a boolean rw indicating if the page was accessed while reading or writing,
  - the state of the architectural registers reg of the simulated target,
  - the index x of the target register whose value is either updated in case the instruction that led to the page fault is a load or used if it is a store,
  - the address at which the fault occurred, $addr$,
  - and finally the address at which the handler will return, $raddr$.

We first call the QEMU `softmmu_helper` (line 2) in an *ad-hoc* mode, asking the helper to only perform the virtual target to virtual host translation, but not the actual read or write in memory. If the address is valid, we need to do a page table walk and provide the handler with

the host address corresponding to the access. The page table walk is clearly an architecture dependent algorithm, but since we use it as provided by the existing target implementation in QEMU, we stay fully retargetable. If the translation occurs without any host fault (line 3), this address will be used to update (`fill_page_table`, line 4) our shadow mapping by making the faulting address translate to the same physical address as the virtual host address given by QEMU (`get_phys_addr`, again line 4).

Otherwise, there are two possible causes: either the access is an IO or it is a page fault. If it is an IO, the handler must not establish a mapping as IOs are handled by executing specific callbacks in QEMU. The $reg[x]$ register value will either be updated using a value from a device register or be used to update the register of the device ($*$ means pointer dereference in the algorithm) and the shadow mapping will be left unmodified (line 12).

In the case of a page fault, the target should be passed the fault and left to decide how to react to the page fault. However, since the code is executed in an exception handler, it is not possible to directly jump to the target exception vector. Indeed, in case of a target fault, QEMU needs to make a `longjmp` to its top level code so as to generate the code for the handler and then execute it. The solution is to rewrite the return address of the handler (line 14) so that the execution will continue in the `softmmu_helper`, the code that an unmodified QEMU would normally use to handle target page faults. This works perfectly since returning from an interrupt handler restores all the values in the target registers reg, the right values needed by the `softmmu_helper` helper to interpret the page fault correctly. From there, QEMU will handle the fault as it usually does, relying on the standard memory translation, not the shadow translation.

Data: oi: softmmu info, pf: page fault info
Result: updated shadow page table or IO access executed

```
1  target_addr ← pf.addr − offset
2  host_addr ← softmmu_helper(oi, target_addr)
3  if is_valid(host_addr) then
       // Update host page table
4      fill_page_table(addr, get_phys_addr(host_addr))
       // Access memory as required
5      if pf.rw = r then
6          pf.reg[x] ← *host_addr
7      else
8          *host_addr ← pf.reg[x]
9      end
10 else
11     if is_io_addr(host_addr) then
12         io_helper(pf.rw, host_addr, pf.reg[x])
13     else
           // Rewrite the code to which the handler will return
14         *pf.raddr ← asm_to_bin("call softmmu_helper")
15     end
16 end
17 return from exception
```

Algorithm 2: Page Fault Handler

Most of the time, the target will react to a page fault by changing its virtual address translation, something that needs to be reflected in our shadow translation. Fortunately, we can rely on the QEMU Virtual TLB, since we share with it the same coherency requirements. Essentially, whenever QEMU flushes its Virtual TLB, we need to also flush our shadow

translation.

This is not optimal, since we are flushing the entire shadow translation, when we could update only individual entries. But it is optimal regarding our objective of keeping QEMU mostly unmodified. Going further would require that we manage flushes at the target level *i.e.* catch the target page table modifications by ourselves. While not extremely challenging in itself, this would lead to an important quantity of other issues. Amongst them, we would have to, for it to be any of any use, manage also the target page walk. Indeed, knowing which page to flush is one thing, but we also need to know precisely its size. QEMU Virtual TLB actually solves its problems by managing only 4K page. Each 4K page is aware of being part of a large page to only a very limited extent, the goal being that if one page is modified, the whole mapping is flushed out of the TLB. This essentially means that we can not directly rely on the Virtual TLB to achieve this.

The other issue is on how many single pages will be flushed. It is to be understood that as we flush our mapping we also have to flush our guest TLB. Flushing the TLB, except for the obvious cost due to future TLB misses, also costs a certain number of cycles in and out of itself. If flushing the whole mapping, this is around one hundred cycles. If flushing per page, it is around four to five hundred cycles. In the case where we would flush a target large page, let's say an ARM one megabyte page, that we would have had no choice but to represent as two hundred fifty-six 4K host pages, we would then spend around one hundred twenty-eight thousand cycles flushing that page. If multiple large pages were to be flushed, the negative performance impact would completely offset any potential gains.

4.5 Conclusion

In the end, this work has managed to accelerate the simulation of target memory accesses in the context of DBT based cross-ISA simulation. To do so, we leverage the facilities provided by Dune framework[BBM+12], to create an embedded target address space into the simulator, QEMU here, address space. Then we use this embedded target address space to executes target memory access directly in generated code, without the Virtual TLB as an intermediary. We couple it with the ability to install our own page fault handlers so as to both fill and synchronise this mapping, and simulate target page faults. The result is a solution where QEMU runs on top of Dune so as to be able to leverage its facilities and accelerate target memory accesses simulation.

This work allows us to get two vectors of speedups: accessing target memory in generated code now only needs a couple of `mov` instructions, and the Virtual TLB limitations are gone, with a basically unlimited ability to store translations. Latency to present a page fault to the simulator (internal or target) should also be about four times faster than [WLW+15], as according to [BBM+12], handling a SIGSEGV fault takes four time the time it takes to handle a Dune page fault. Compared to [SWF16], we also do not have to worry about communication between the different parts of the simulator since the whole simulator is running on top of a VCPU.

There are however still a few problems with this idea. HAV is not a fully standard mechanisms, each architecture, including sometimes those using the same ISA, implements its own version of it. This is a first worry for portability. Dune itself is not a trivial application. Porting it to another architecture, especially since it depends on HAV support directly, is bound to be complicated. This essentially means that while we retained QEMU frontend retargetability, it seems that we lost at least partially this ability on backends.

Secondly, Dune is no longer supported and does expose bugs in a quite a few situations,

which is not surprising with a very complex program that is a simulator (many threads, dynamically generated code with hand crafter contexts...), and while already quite impressive is still only an academic realization. One of the result is that the ability to debug has also been largely reduced.

Another limitation is that the use of Dune requires QEMU to be compiled in static mode. While not being a problem in itself, compiling QEMU in static mode disables many of its features, such as its support for graphical interfaces and several target devices. This is a limitation due to the current state of the implementation of QEMU and bears no consequence on the validity of our research results whose objective is to exhibit memory management issues and potential solutions for better simulation performance.

Finally, HAV may not always be available. If the workload has to be run on a distant virtual machine for example, the native HAV will already be in use. This means that nested HAV will be necessary for Dune, this in turn means potentially reduced performance.

This work also has some solution specific limitations. As we just discussed, our management of the shadow translation is not optimal since we are flushing the entire translation when we really should only flush the Virtual TLB of the vCPU in Dune. But there is a trade-off to a finer-grain coherency. Future work needs to assess the gain in performance to be expected from a finer grain coherency versus the consequences of modifying QEMU further, diving in the details of the software MMU simulation of QEMU. In particular, we would need to assess the portability of such modifications, across versions of QEMU, but also across the different targets supported by QEMU.

A more severe limitation is that we are limited to 32-bit targets. This is mainly due to an address space collision problem. Basically, we are embedding the target address space into the simulator address space. Now, if we imagine that we have a 64-bit target, directly mapping the target address space would clearly be a problem. We would overlap with the simulator address space, which is not possible. However, one direct solution to this problem, with an extended use of Dune would be to create a second address space just for the target.

This would not be fully straightforward to implement as we would also need to map the generated code as well as entry points to still be able to call QEMU helpers in this address space. However, we can imagine that we could support up to 47 bits wide target addresses effortlessly, and considering AArch64 (the 64-bit ARM ISA) is still using mostly only 42 bits, this would be highly sufficient for all currents, short-term and mid-term embedded uses. This should also even fit for nowadays smartphones probably. This could even be naturally extended to 56 bits too with Intel 5 level page table support, which is already designed while not yet available in products. In which case, any ARM device could be supported for the foreseeable future.

Overall, the HAV related limits of this solution do lead us to explore another path. That path is to explore an analog solution without the use of HAV. Its requirement would be in particular to expose custom fault handling so as to keep fast fault handling. The goal being to keep fault handler latency much lower than in a mmap based solution such as [WLW+15].

Chapter 5

Cross-ISA simulation using a Linux kernel module and host page tables

5.1 Introduction

In this chapter, we try to bring an answer to the questions "What is the most efficient method to implement memory simulation acceleration schemes?" and "Are the designs we choose to accelerate target memory accesses simulation implementation specific, *i.e.* tied to the underlying OS or framework we use, or can they be reused atop any of those?" as well as still adding in to the answer to the first question "What schemes may be devised so as to accelerate target memory accesses simulation?".

In the previous chapter we have presented a solution based on hardware assisted virtualization and the Dune framework to accelerate the simulation of memory accesses by a target CPU. This solution was, in fact, mainly based on embedding the target address space into the address space of the simulator, thanks to the hardware support for a second level page table. It also provided a fairly fast page fault handling mechanism, since it allowed us to manipulate this second level of page table directly from the user application, keeping the host OS away from the page fault handling path. However, we also discussed the problems brought by the use of HAV and Dune, such as interoperability and debugging complexity.

In this chapter, we propose a solution similar in spirit, but which does not depend on HAV or Dune. Just as for the Dune based solution, we aim at embedding the target address space into the simulator one. However, we only manipulate the host page table to do so, and never use the second level address translation as we did before.

Quite obviously, the simulator, now that it is again a standard unix user process, can no longer directly manipulate its page table. To gain access to the host page table, we will thus rely on a Linux Kernel Module (LKM). We are aware that a solution based on `mmap` is possible, as detailed in [WLW+15], however we want to explore a new path with an LKM so as to try to obtain a page fault handling latency as close as possible to that of the Dune solution. We are also aware of the work of [CWH+14] that tries to use an LKM, but it is basically only a reimplementation of `mmap` without really any addition to it. In their solution, accesses to the embedded target address space will require synchronizations, which will be done through page faults. However, these page faults go through the standard Unix SIGSEGV mechanism. As such, it must go through the whole Linux page fault handling mechanism, plus the relatively costly UNIX signal mechanism. In order to do better than a basic `mmap`, our LKM provides a custom page fault path to avoid the signal problems. In its proof of concept

form it also exposes the ability to manage, in kernel, a part of the target memory emulation, thus making the fault handling path shorter. Our main hope with this method is to obtain lower page fault handling overhead compared to [CWH+14] and [WLW+15] which basically follow the standard Linux page fault path.

This solution was designed to be general, but as it is quite deeply embedded in both the operating system and dynamic binary translation engine, we detail it and implement it once again with QEMU and with a Linux kernel module.

5.2 Design of our Solution

The goal of our solution is to strive toward native performance for simulated memory accesses, translating as directly as possible target memory accesses into host memory accesses. With an Arm target and an Intel x64 host, this means translating a `ldr` or a `str` instruction into a single `mov` instruction. Our design relies on two key points: mapping the target virtual space in a region of the host address space and controlling fully the mapping of that region via the host memory management unit (MMU).

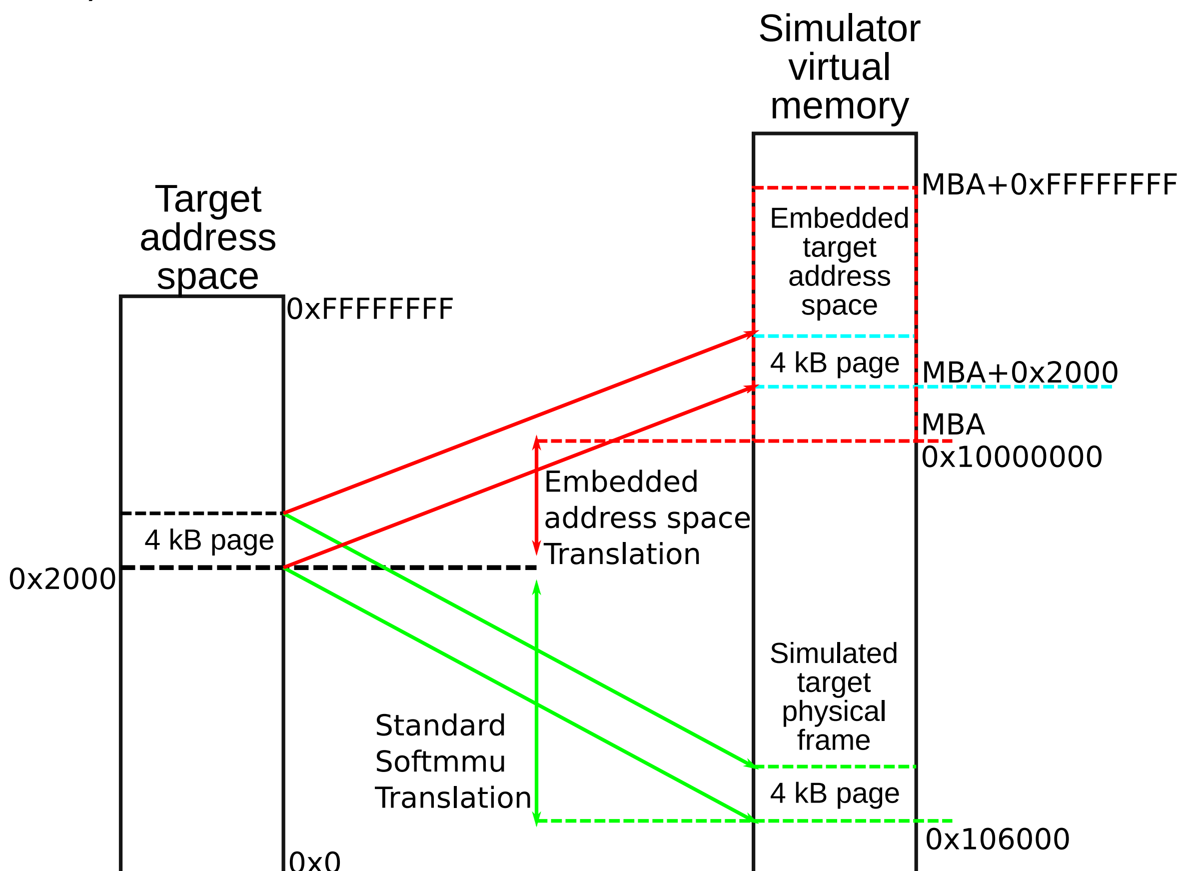


Figure 5.1: Embedding target address space (a)

Figure 5.1 depicts the mapping of the target address space in a region of the host address space from the simulator perspective. First of all, the usual QEMU translation, represented by green lines in the figure, is still present. We see the example of a 4 KB target page at target

address `0x2000` being translated through the usual Virtual TLB/`softmmu_helper` algorithm to a 4KB host virtual page at address `0x106000` representing the target physical frame. This translation remains the one in use by the various helpers used by QEMU for the simulation.

Parallel to this, a 4 GB contiguous virtual memory area is reserved in the simulator address space. This is the space that will be used to perform the mapping which will be used by the "inline" memory accesses, *i.e.* the ones in the generated code that do not use helpers. This mapping, in this example, resides at address `0x10000000` which we shall call `MBA`, for Mapping Base Address. There, the aforementioned target page at target address `0x2000` will be mapped at host virtual address `MBA+0x2000`. This mapping will then be usable at runtime, directly in inline code, through a simple host "`mov [target_addr+MBA], src/target`" instruction to execute any memory access on that page. What this means is that we now have the possibility to translate target memory accesses into essentially one host memory access, with possibly an arithmetic instruction to add the MBA to the target address.

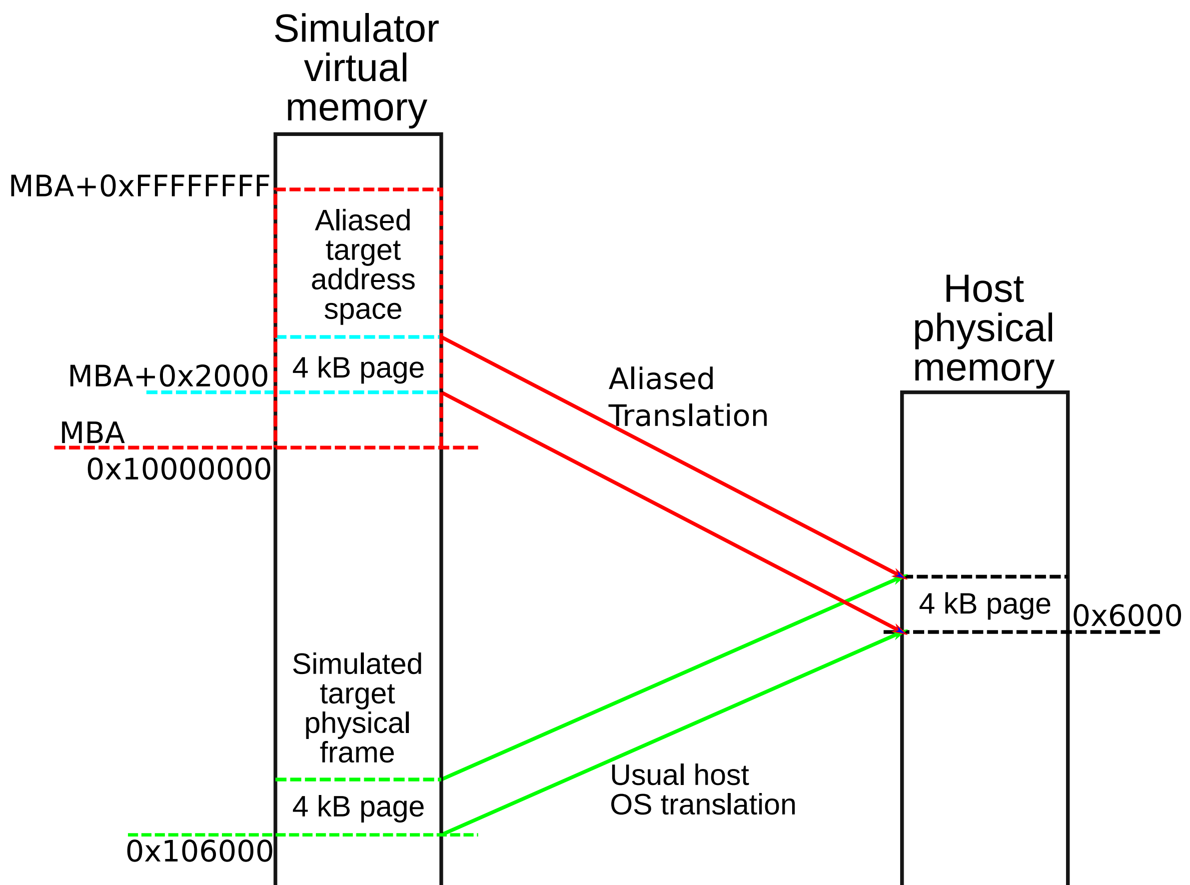


Figure 5.2: Embedding target address space (b)

The way this mapping actually functions is described in Figure 5.2. If we look first at the target physical frame, as seen by the simulator, we see it is now mapped at host physical address `0x6000`, this address being decided arbitrarily by the host OS. Any modifications to host physical address `0x6000` will then be seen by the simulator, and any fetch will return the value seen by the simulator. To implement our mapping, we then have the host MMU

alias our representation of target page, at address MBA+0x2000, to the host physical address 0x6000, which is the same as the simulator representation of the target physical frame. To find this address, we simply call QEMU standard code at the time of the first access to MBA+0x2000. Essentially, this means that now, any host access to either virtual address 0x106000 or MBA+0x2000 will be strictly equivalent. Loads will return the same value, and stores will update the same physical address, therefore later loads at both virtual addresses will return the updated value. However one important difference is made between these pages in terms of how they are accessed. Indeed, 0x106000, which is used by the simulator and never directly by the host code resulting from a translation of the target code, is always protected by runtime checks. As such, it can freely by given read/write access rights at any time to simplify simulation. On the other hand, MBA+0x2000 will be accessed inline, without runtime checks, by the target simulation code. As such it needs to reflect the same protections as the target page that includes target address 0x2000, otherwise, illegal target accesses may succeed even though they should trap, thus making the simulation incorrect. With appropriate access rights on the other hand, target illegal accesses, such as writes on read-only pages will be caught and the fault will be transmitted correctly to the target, thus correctly emulating target behavior.

Finally, it must be remarked that we always speak of target addresses, target address space and not target virtual addresses or target virtual address space. This is due to the fact that our mapping is actually totally oblivious to the type of target address space in use, the only information we need is its size. To obtain the actual host physical address we need to map to, and target access rights, we simply reuse QEMU softmmu_helper, keeping our own code oblivious to those details. This means that we are also completely target ISA independent as far as the mapping goes. Indeed, any and all target specificities, such as exact address space size or type (physical or virtual) will be covered by QEMU softmmu code already. We do not make any virtual to physical translation by ourselves, and we never read or write any target register. And the addresses we manipulate ourselves are host virtual addresses representing target physical frames, which are by definition completely independent of the target.

This design is quite straightforward and can be expected to exhibit near native performance, but it raises a few questions. First, the control of the host MMU is privileged on all general purpose operating systems, but fortunately, most operating systems are also extensible with user-defined modules. So we will use a kernel module to manipulate the host MMU, more on this in Section 5.3. Second, the respect of the target semantics (access size, sign extension, endianness) may be sometimes tricky. Indeed, while in the inline code most details of the semantics are already handled by QEMU (access size, sign extension, endianness, etc.), in our version of the generated code, and in our fault handler, those are our responsibility. Most of the times, however, our solution works as described above, with only a single mov instruction, chosen to fit the sign extension, size, and other constraints, plus an arithmetic instruction per translated target ldr or str instruction. But depending on the target and host processors, the translated snippet of code might require more instructions than just a single mov instruction. Simulating different access sizes or sign extension might be tricky, but it can usually be done by choosing the right sequence of host instructions. Simulating traps, like page faults and protection faults, may prove more difficult if the target MMU and the host MMU do not share similar protections and traps, which is fortunately also rare. If the target and host do not share the same endianness, we will need a few more instructions to shuffle bytes. In most cases, the host processor has an instruction to shuffle bytes, such as the Intel SIMD SSE3 instruction pshufb (packed shuffle bytes) or bswap x86 instruction, tbl on AArch64 or vperm on Power.

We shall however note that the handling of the above tricky cases is not specific to our solution, all DBT techniques face a similar extra overhead in order to handle these cases, including baseline QEMU. In contrast, simulating privilege levels in processors require special attention in our solution. Indeed, we cannot rely on the host MMU to do the checks related to the processor mode (kernel or user), since the simulator being a regular process, it executes in user mode. We could think of using the privilege protection bit in the MMU on kernel pages, but the simulator would no longer be allowed to access these pages since it is a user-level process. Removing the privilege protection bit would not work either because it would mean that a privileged page would not be protected from unprivileged accesses.

To make matters worse, there are some other cases where our approach might suffer from high communication overheads between our kernel module and the user-land simulator. The emulation of hardware devices is always a delicate and costly endeavor, but our approach makes it worse. The problem comes from the fact that the interface of hardware devices is memory-mapped, based on memory-mapped input-output hardware registers called MMIO registers. This means that device drivers are reading and writing such MMIO registers and each such memory accesses must be emulated, based on the emulated hardware device. This means that our approach will need to trap upon each read or write instruction, percolating the trap up to the simulator. Given the expected overhead of such a mechanism, the approach may not seem practical after all.

Fortunately, there is an absolute win-win solution. The idea is to optimize memory accesses when it will be efficient to do so and use the standard approach otherwise. In other words, we will gain when we can and lose nothing otherwise. The problem is that it is a decision that we need to take at translation time. Indeed, the idea would be for the simulator to translate a memory access as it would normally do only if that memory access is expected to be trapping. Otherwise, the simulator would generate code with a single memory access with the `MBA` offset added. Technically, it is quite easy to translate a code block either one way or the other, or even considering each memory access individually.

The hard part is to know statically when to translate as always or when to translate using our approach. In the general case, the corresponding knowledge is not available statically. But it so happens that we can leverage the processor mode (user or kernel) to decide. Indeed, accesses to hardware MMIO registers may only be done in kernel/supervisor mode. As such, all device drivers execute in privileged mode. Other problematic behaviors for us, such as frequent TLB flushes also only happen in kernel code since flushing the TLB may only be done in privileged mode. We then can say that essentially, problematic behavior is localized in the kernel code, running in privileged mode.

When the simulator translates a code block, it knows if the processor is in user mode or kernel mode because it manages all the target registers and architected state, and it catches any mode switches. There is then a simple and efficient way to decide whether to use QEMU standard translation, or ours. In kernel mode, the translation will be done as usual. In user mode, the translation will rely on our approach.

5.3 QEMU/Linux Implementation

This section describes our prototype based on Linux and QEMU. Overall, we were able to leave QEMU mostly untouched, with only a few modifications to the dynamic binary translation and the addition of our mapping of the target address space via the use of a `map ioctl`. Regarding Linux, we add our own Linux kernel module to handle that `map`.

5.3.1 Kernel module

Our module creates a virtual memory region to host the mapping of the target address space. Once the region is created, memory-related exceptions happening in that region induce a communication between our module and QEMU. MMU exceptions are percolated up to QEMU, allowing QEMU to decide how they should be handled. This usually means that QEMU will ask our module to make some MMU modifications.

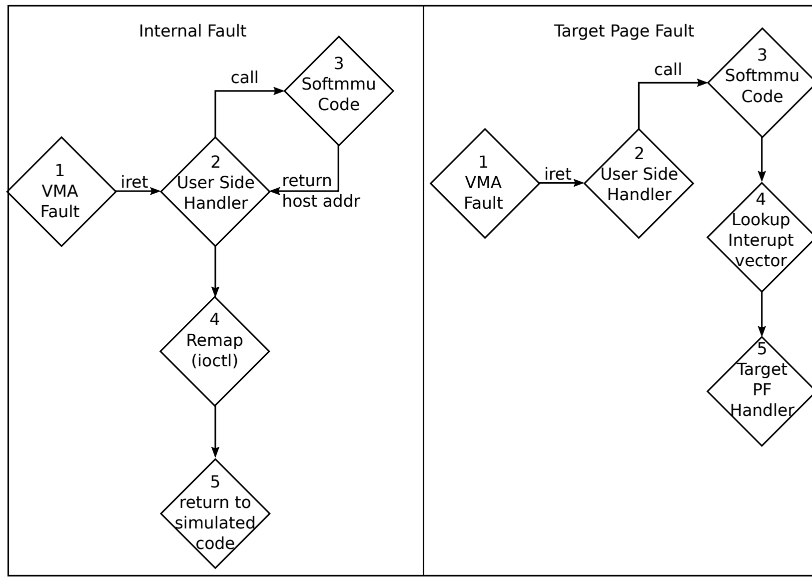


Figure 5.3: Overview of the implementation

To create the virtual memory region, our module relies on the Linux mechanism called Virtual Memory Area (VMA). A VMA is a contiguous range of virtual memory addresses in a process for which one may define a specific exception handler. That handler is invoked by Linux whenever an MMU-related exception occurs as a result of a memory access within the address range of the VMA. In our implementation, our exception handler will percolate the call up to a user-land QEMU *handler*. It is important to note that Linux invokes the exception handler on the thread that attempted the memory access that induced the MMU-related exception. In other words, the handler executes in the environment of the QEMU thread that executes the translated target code.

To percolate the handling of the exception, our exception handler manipulates the Linux context so that Linux will not resume the execution of the target code at the memory access instruction that caused the exception, but at one of specific handler that we added to QEMU. This process can be seen in Figure 5.4. The exception handler is called when a memory access instruction, in QEMU code cache, traps due to our mapping not yet being filled. The handler first searches the VMA to which the faulty address belongs. Then, we retrieve the user side handler address and store it in place of the original RIP value in the exception frame. Finally, we copy the exception frame to transfer it as a user side handler argument and return from exception.

That QEMU handler will interact with the rest of QEMU to determine what needs to be done about this exception. Usually, this means that our QEMU handler will ask our kernel module to modify the MMU setting of the corresponding VMA, via `remap` ioctls. Upon the completion of our QEMU handler, two outcomes are possible. The first outcome happens

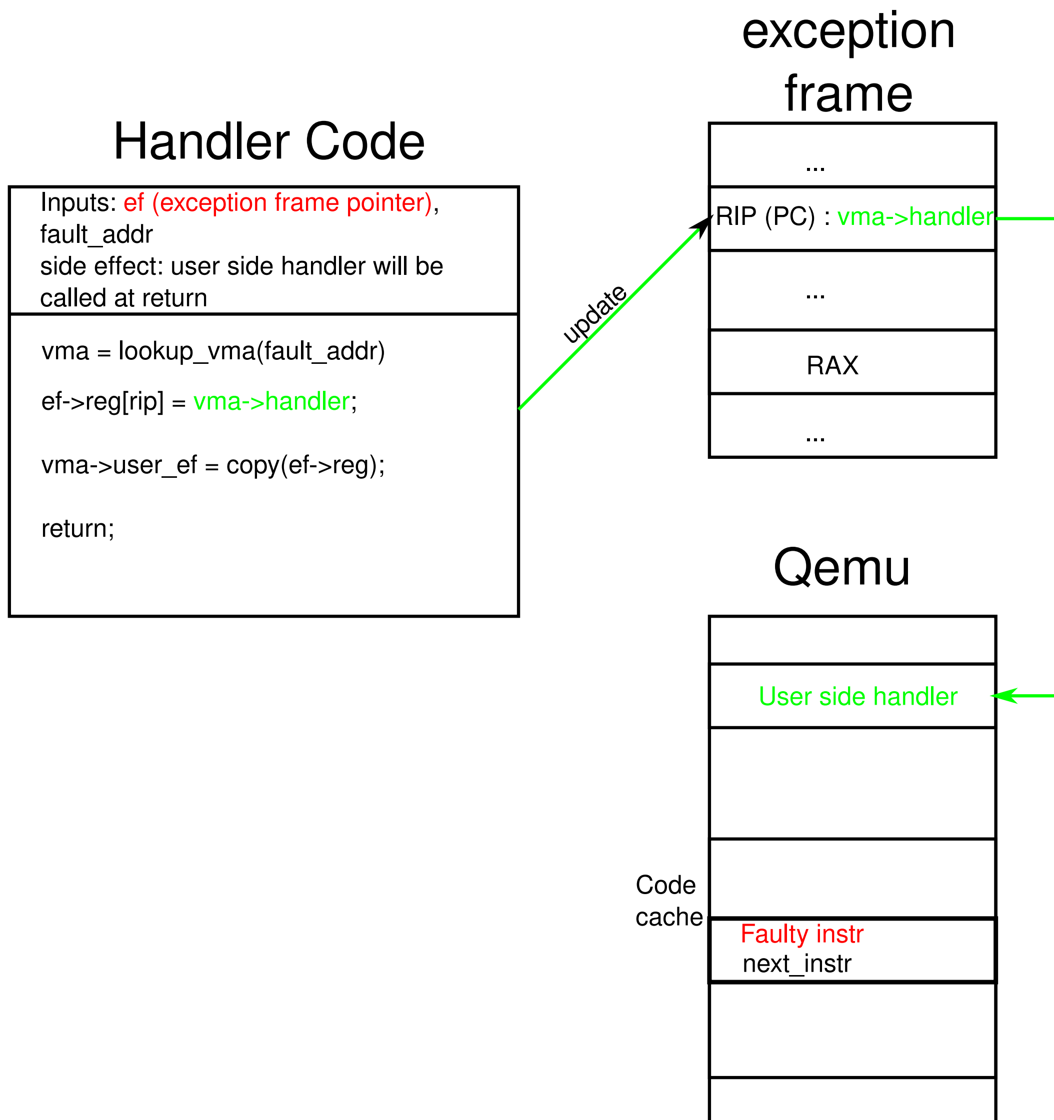


Figure 5.4: Update of the exception return address

when the situation that caused the trap has been resolved. In this case, the execution should resume at the instruction that caused the exception that triggered this whole process. This is what happens at startup, since the target executes in physical memory with no MMU enabled. In that case, the QEMU handler establishes a first mapping within the VMA. Indeed, as explained Section 2.2.3, target physical pages are allocated by QEMU, as page-aligned page-sized frames, using a user-level memory allocator within the QEMU process. As such, a frame has a virtual address in the QEMU process and that virtual address can be translated to the address of a physical page in the Linux kernel. That frame virtual address is simply forwarded back to the module via a `remap ioctl`. Our module then simply aliases that corresponding physical page from the VMA. Consequently, the VMA mapping fills up as the target execution accesses memory addresses. The second outcome happens when the situation that caused the MMU exception could not be handled by our QEMU handler. At startup, this happens when the memory access is invalid, typically at an address that does not correspond to any physical memory or MMIO registers in the target memory map. In

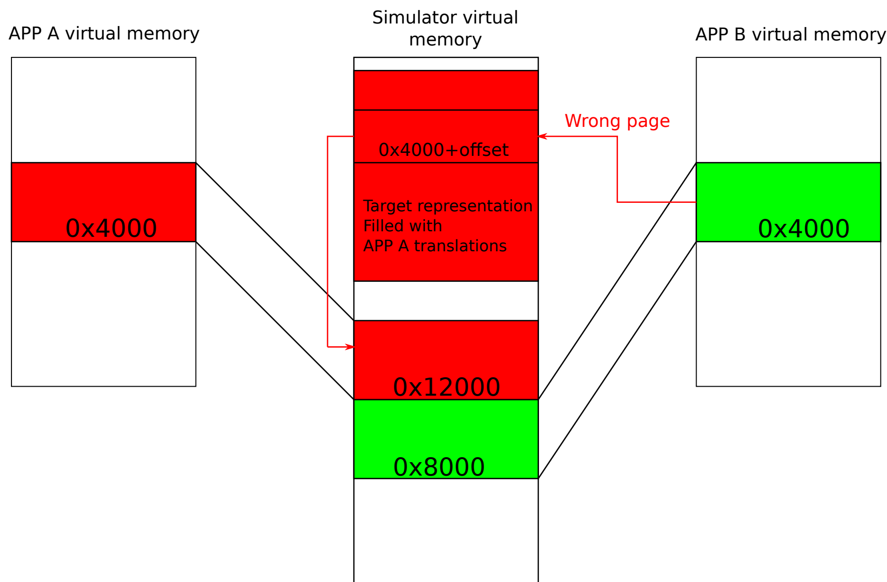


Figure 5.5: Handling context switches

that case, we delegate to QEMU the emulation of the corresponding trap.

So we just discussed the case where the target has its MMU turned off, but of course, the target will enable its MMU at some point, switching from using physical addresses to using virtual addresses. This changes very little from our point of view, the overall processing of MMU exceptions continues almost identically. The only difference happens in our QEMU handler, with two distinct situations that are now possible. The MMU exception could be an internal affair, meaning that our VMA is missing a mapping, but the target has set up its MMU with a mapping. This means that our QEMU handler knows how to resolve the missing mapping and it will ask our module to update the VMA mapping accordingly, again via a `remap ioctl` as explained above. This situation is illustrated on the left-hand side of Figure 5.3. It may happen however that our QEMU handler discovers that the target MMU is missing the mapping, from the target perspective. In that case, the target is expecting a page fault at that address, so we delegate again to the regular QEMU code the emulation of the corresponding trap, executing the adequate target handler. This situation is shown on the right-hand side of Figure 5.3.

That execution of the target handler is likely to change the target MMU setup, as a result of resolving the MMU exception. Of course, these changes need to be reflected in our VMA mappings. But it is important to realize that this is only one of the possible situations where we must keep our VMA mappings in sync with the MMU manipulations of the emulated target. We must consider several MMU-related management operations:

- Target enabling and disabling the MMU,
- Target applying changes to the current page table,
- Target switching page tables.

When the target enables or disables its MMU, our entire VMA mapping needs to be flushed. The same is true when the target context switches, changing the current page table.

Assuming the emulated target architecture executes an operating system on which two processes, A and B, are running, Figure 5.5 depicts what would happen at the moment of the

context switch from A to B if we did not flush our entire mapping. In both processes, the virtual address `0x4000` is valid. Upon the context switch, assuming the address `0x4000` has already been mapped while executing process A, any access by process B would result in a successful, but erroneous, access from the perspective of host MMU. If we invalidate our mapping, the access would be handled as explained earlier, percolating the MMU exceptions up to QEMU and the establishment of the correct mapping.

So the question becomes the following: when should we invalidate our mapping? In QEMU, this can be tracked since QEMU already implements a Virtual TLB, which faces the exact same question. We therefore modified QEMU so that whenever QEMU flushes its Virtual TLB, we ask our kernel module to flush its mapping. We do so through an `ioctl`. It is interesting to note that QEMU flushes its Virtual TLB for any relevant MMU change, including when the target makes very localized changes to the current page table, such as changing page protections, unmapping a page, or remapping a page. The good news is therefore that it is the correct hook, we always invalidate our mapping when we need to, maintaining a correct emulation of the target execution.

The bad news is that we invalidate our entire mapping even if the target only changes the translation of a single page or its access rights, which incurs unnecessary overheads. We tried to implement a more focused invalidation to mitigate these overheads. We were able to distinguish when the target adds new translations to the page table, which is a change that does not require to invalidate our mapping. We also did not invalidate our mapping when the target augments the access rights to a page. We take care of this situation lazily when handling the related MMU exception, realizing that the access rights needs to be changed and the access was legal. Other than that, modifying QEMU in order to have precise information about what should be invalidated in our mapping proved quite difficult. It is one of the facets of our design that is hard to retroactively fit in an existing production-grade simulator such as QEMU, and overall, we are still invalidating too often and too much.

Our performance numbers could be improved with a deeper integration of our mechanism in QEMU. Until then, we need an implementation of our invalidation that is fast. Unfortunately, the invalidation of the entire mapping of a VMA by Linux is expensive, it is essentially a loop over all the page entries of the entire page table. For small non-sparse target address spaces, this is not a problem. For large or sparse address spaces, it may become a source of a noticeable overhead. To avoid this problem, our module maintains its own list of page entries that are currently mapped. Invalidation is therefore about scanning that list, asking Linux to invalidate individual pages. Future work would require to be able to track individual changes by QEMU, the intuition would be to log such changes so that they can be applied by our module when QEMU flushes its Virtual TLB.

5.3.2 Dynamic binary translation

With the mapping of the target address space in place, the dynamic binary translation of memory accesses can now be changed. Figure 5.6 contrasts the original binary translation in QEMU and ours, for a simple memory access. In the figure, the memory access is a `ldr` instruction, but considering a `str` instruction would be essentially identical: for both instructions QEMU translates memory accesses with a combination of a fast path, which does a direct computation, and a slow path that calls a helper.

As depicted on the left-hand side of Figure 5.6, the fast path is inlined at the memory access site in the code block. It contains multiple instructions, one branch, and multiple memory accesses. The instructions lines 0 and 1 are about computing the effective address

at which the `ldr` takes place into a register (`ebp` in this case). Then, we have the address translation, composed of the inlined fast path and the branch to the slow path code (line 10), in case the fast path fails (check of the Virtual TLB done line 8). If it is there, the memory access can be done (line 12). Line 13 is the start of the translation of the instruction which follows the `ldr`, and it is the trampoline return target.

Figure 5.6: Fast path in case of standard QEMU code (above) and our solution (below).

If the translation is not available in the Virtual TLB, the code branches to the slow path trampoline code that calls the QEMU helper. This helper will use the QEMU `softmmu_helper` in order to perform the translation, update the Virtual TLB, and then execute the memory access. The code of the slow path is essentially about preparing the value of the parameters that are passed to the helper function. One such parameter deserves our attention, it is the *operation index* (noted `oi`) at line 2 of the slow path trampoline code. The operation index in QEMU is a description of the memory access that is generated at translation time by QEMU. It contains the following information: read access or write access, endianness, sign extension, and size in number of bytes. That operation index will be used by the helper to check if the memory access is valid on the one hand, and on the other hand to know how to actually perform the memory access, writing or reading a value to or from the emulated physical memory of the target.

On the right-hand side of Figure 5.6, we present the code that our approach generates for the same memory access, when the processor is in user mode. As one can see, we have not fully reached our goal—translate one `ldr` or `str` instruction to a single `mov` instruction—but we are quite close. We have eliminated both fast and slow paths, and we only have two extra instructions. The first extra instruction, line 3, consists of loading the `MBA` offset in a register, so that we can afterwards compute the memory access address as explained Section 5.2. The second instruction, line 2, loads the operation index that concerns this memory access in an other register. We do this in case the MMU shall raise an exception. That way, the exception handler in our kernel module can find the operation index in the register value saved by the Linux trap mechanism. This is necessary because we need to percolate that operation index up to our QEMU handler.

```
Data: pf: page fault info, include register values
Result: update embedded target mapping or forward fault to simulator
1  target_addr ← pf.addr − MBA
   // retrieve architected state, pointer is in r14 in Qemu
2  arch_state ← pf.reg[14]
   // retrieve arguments we passed to ourselves in generated code
3  oi ← pf.reg[rsi]
4  host_virtual_address ← softmmu(arch_state, oi, target_addr)
5  if is_valid(host_virtual_address) then
   |  // IOCTL to signal the LKM to alias the address
6  |  ioctl(IOCTL_REMAP, pf.addr, host_virtual_address)
7  else
   |  // Handle real fault or io
8  |  if is_io_addr(host_virtual_address) then
   |  |  // execute Qemu io callback
9  |  |  io_callback(host_virtual_address)
10 |  end
11 |  if is_fault(host_virtual_address) then
   |  |  // In case of target faults, update cpu arch state and go back to
   |  |     code generation, the target handler will be executed
12 |  |  arch_state.fault ← true
13 |  |  longjmp_to_code_gen
14 |  end
15 end
   // restore registers as they were before page fault, and go back to
      generated code, but to the next instruction
16 pf.reg[pc]+ = 8
17 restore_regs_and_return_to_code
```

Algorithm 3: User side page fault handler

The pseudo-code of the handler is given in Algorithm 3. As a matter of fact, as we can see there, our kernel module passes the same arguments to our QEMU handler as the arguments passed to the regular QEMU slow-path helper. This makes perfect sense since our handler will interact with the rest of QEMU in pretty much the same way as the original slow-path helper. The difference is not in the translation of the address, as it is done by the `softmmu_helper` identically in the two cases, and is represented by the call to `softmmu_helper` at line 4. Access

is also done identically by the `softmmu_helper`. Our QEMU handler does not however update the Virtual TLB since it is no longer used by the code translated in user mode. Instead, our handler communicates back with our kernel module, via a `remap ioctl` at line 6 in algorithm, in order to update the host MMU translation for the translated address. The handler finally resumes the emulation at the next instruction in the translated target code (lines 16 and 17 in the algorithm).

This last point is a bit challenging on a CISC host like an Intel x64 processor. Indeed, QEMU generates different `mov` instructions that lead to different instruction lengths, *i.e.* instructions are coded with a variable numbers of bytes. We solved this problem by padding all `mov` instructions to 8 bytes, generating no-ops for any extra byte. Most of the time, this padding is small since a vast majority of `mov` instructions that QEMU generates are between 6 and 8 bytes long.

5.4 Handling Internal Faults at Kernel Level

One problem we might see with the solution as we just described it is that handling faults is an important performance bottleneck. This is mainly due to the fact that, for internal faults, we are making round trips between kernel and user space handlers. However, if the fault is internal, QEMU (and the target) do not need to be aware of this fault. As such, the need to reuse QEMU code unconditionally should be possible to lift. However, three main obstacles exist on that road. First, we need to be able to make target and shadow page walks in kernel. Second, we need to preserve QEMU vision of memory accesses *e.g.* we need, for example, to forward to QEMU the information that a write access to code happened. Finally, we need, when the fault can not be handled directly by the LKM, to percolate it to the simulator.

Let us now detail the implementation of our proof of concept work on using an in kernel internal fault handler. As we have seen in Chapter 2.2.3, to emulate target memory accesses we need to have access to two essential data structures, target page tables, and a shadow page table. Indeed, while the target page tables give us access to target virtual to target physical address translation, we can not directly use it. We need to know where, in the simulator address space, does each target physical frames pointed by the target physical addresses we just computed lie in simulator address space. This is what the shadow page table provides, a target physical address to host virtual address translation. As such, to handle internal faults in kernel, we need to make those two structures available to our module.

The first one we actually need is the shadow page table, as without it, the target page table walk can not be made. However, making calls to user side code is not possible in supervisor mode, mainly for security reasons. As such, we need to have our own shadow page walk code. To make it feasible in kernel, with limited stack size in an exception handler, the shadow walk should be as small (and fast) as possible. Ideally, the shadow page table should pretty much be an array or at most a simple hash map. QEMU code however does not have this restriction, as such, its shadow page table is implemented in the form of a fairly complex structure based on section and block view of RAM.

To work around this problem, at simulator boot, we reconstruct a shadow page table in the form of a simple array. The advantage is that this structure can be accessed in $O(1)$, and since we never need to modify it, the refill cost is irrelevant to us. In term of memory, it is also largely acceptable, we are making one 8-byte entry per page. With each page being 4 KB large, or 2^{12} bytes, if we consider a 2^{32} bytes large target address space, this means that we need 2^{20} entries of 8 bytes. This means that we need a 8 MB array for the shadow page table using this totally trivial strategy. This would obviously not scale for a much larger

address space, but in this case a hashmap filled as physical memory is accessed would be a reasonable option.

Once we have the shadow page table, we need to be able to walk the target page table as well. This step, once again, can not be done directly using user code. However, we can solve the problem by copying it. Indeed, the code in itself is relatively simple. It has few dependencies (mainly the target page table pointer and state register) and only consists of a sequence of physical address dereferences, hence the accesses may be made using our shadow page table. Thanks to this, we can copy QEMU target page walk in the module with only minor modifications. With this and the shadow page table we can now, in the module, make a target virtual address to host virtual address translation.

```
    Data: pf: page fault info, include register values
    Result: update embedded target mapping or forward fault to simulator
 1  target_addr ← pf.addr − MBA
 2  target_phys_addr ← target_virt_to_phys(target_addr)
 3  if is_valid(target_phys_addr) then
        // Lookup shadow
 4      host_virtual ← target_phys_to_host_virt(target_phys_addr)
        // If the host virtual address exists, alias pf.addr to host_virtual
 5      if host_virtual != NULL then
 6          alias(pf.addr, host_virtual)
 7      end
 8  else
        // set return address to our user side handler, and return
 9      pf.reg[pc] ← user_side_handler
10      return_from_interrupt
11  end
12  return_from_exception
```

Algorithm 4: Module Side page fault handler

We can now modify the module side of the page fault handler, which originally pretty much just called the user side handler, to behave as Algorithm 4. What we see there is that a new step has been added prior to forwarding the fault to the simulator. In fact, if this step succeed, we no longer forward the fault. This step first consists in attempting to make the translation from the target virtual address we just attempted to access to the host virtual address corresponding to the simulated physical frame it should point to. If this is successful, then the faulty address will be aliased to the same host physical address as the host virtual address we just found. Otherwise, we simply go back to the slow path, *e.g.* we forward the fault to the simulator.

As can be seen on lines 4 and 5, there can be two cases where the translation fails. First, the translation from target virtual to target physical fails ($target_phys_addr$ is invalid). In that case, we have a target page fault. This obviously needs to be forwarded to the target so that it may handle it. The other case is when the shadow translation fails ($host_virtual$ is NULL). This can have two inner causes as well. Either the access is an MMIO, in which case no simulated physical frame exists and a callback has to be triggered instead. Or it is a code page, in which case the simulator protected it (and not the target) so that any write to it triggers a callback to invalidate potential translations of this code in the code cache.

Overall this technique adds a new trade-off, but one that, as will be seen in experiment

chapter 6.4.4, is positive in the end. Basically, whenever an internal fault happens, we will now be able to handle it much faster, as we avoid not only a round trip between kernel and user side handler, but also a second registers state restoration and full `softmmu_helper` run. However, the cost is that in case we fail to manage the fault in kernel, we will have added a larger slowdown to the translation process. Considering that the traditional round trip, using an `ioctl`, already costs around 1000 cycles, the probability of a gain is much larger. This is once again confirmed in the experiment chapter.

However, another trade-off has to be done here, and that is on direct retargetability. Since we have to copy page walk code into QEMU, this means that we need to have one page walk version per QEMU target architecture in the module. While copying and adapting the code could be automated by replacing or stubing QEMU function call in the page walk, this remain a supplementary maintenance burden. However, considering the performance improvement shown by this method, we believe it remains an acceptable trade-off.

5.5 Conclusion

In the end, we managed to implement a solution analog to that of the previous chapter but without the use of HAV. We once again embed the target address space into the simulator address space. However, the simulator is no longer being run in supervisor mode, we now use the support of a Linux kernel module to handle the host page table modifications The simulator we used is still QEMU and in particular, the modifications made to the simulator are fairly similar to the ones made for the Dune based solution. This allows us to potentially share a fair part of the implementation between both strategies.

As a result, we defined a new design that provides a better ability to debug the simulator, as well as the ability to use any QEMU backend. We also enhanced the design so as to handle, whenever it is possible, faults in the kernel module so as to avoid costly communications with the simulator and reduce faults overhead to the minimum. Thanks to this solution, we can hope to handle faults in a way that is closer to the speed of the Dune based version.

We have also shown in this chapter that part of the memory accesses are actually still fairly slow to simulate using our method. Those mainly are IOs, protected memory accesses as well as memory accesses in the case the TLB is often flushed. All those cases are however mainly characteristic of one type of application: system code. To avoid unnecessary performance loss in those cases, we propose to reuse QEMU standard `softmmu_helper`/Virtual TLB code path to handle system code. The result is a further enhancement to global performance of the solution as will be seen in the experiment chapter.

For all its good, this method still has a few limitations. The main one, just as with the Dune based solution, is that it does not yet handle 64 bit targets. However, this solution is already made to work with weak assumptions on the embedded target address space. In particular, we can work with multiple virtual memory area in parallel This means that in future works we could extend this solution to lazily allocate sub parts of the target address space, with each sub part being assigned a vma. Then, at runtime, we could make one more indirection to access MBA, which would no longer be unique but depend on the target address space sub part.

For now, we also have only partial multi-threading support. Not the whole process is thread safe, which causes crashes in multiple case. The way to handle multi-threading is otherwise fairly straightforward, we allocate (at least) one vma per threads, and each thread can have its own embedded target address space, with each of them handling its faults in its own thread.

Finally, our proof of concept work on handling internal faults directly in the kernel module only manages read accesses. This is due to the need for us to be able to communicate writes to some specific areas, including code, to the simulator. This is currently relatively complicated as we also need to retrieve information from the simulator, at runtime, and update our structures in kernel simultaneously.

For the two first issues, in fact, what we mainly see is that QEMU is not really fitted toward the use we want to make. This bring us to envisage, as a future work, a simulator that would be totally fit, in particular in the way the architected state is handled, for us with our LKM, or even directly with HAV.

Chapter 6

Experimentations

6.1 Introduction

So far we have seen in Chapters 4 and 5 two methods to accelerate memory accesses simulation in Dynamic Binary Translation. Those two methods use a relatively similar design but with a fairly different implementation. The potential challenges and performance bottlenecks are also not the same. One is based on HAV and runs with a partial bare bone mode, while the other relies on a Linux kernel module for low level details, which requires taking special care of the communication between the simulator in user mode and the module.

In this chapter, we first evaluate the present state of QEMU, with in particular the breakdown of the time spent in its subparts during simulation. Then, we present the memory access simulation acceleration mechanisms that are currently implemented in DBT engines and specifically in QEMU. In particular, we show how important they are, and the key role played by memory accesses in term of simulation time.

In a second time, we present the evaluation of both contributions we made in this thesis. We first evaluate separately the performance of both, and for each provide a breakdown of their own potential bottlenecks. Then, we compare them directly and conclude on which method seems to be the most efficient currently. The evaluation will mostly be about the speedups they provide as well as their potential communication problem, for the first one with the host, and for the second one with its companion Linux kernel module.

6.1.1 Benchmark set

The programs we use for our experiments are described Table 6.1. All "computational" programs are from the polybench suite [PY16]. They are split in three categories, "Long" for those with a run time of more than 100 seconds, "Medium" for those with a run time of more than 20 seconds and "Short" for those with a run time of less than 20 seconds. This breakdown was made to fit the point were our solution has less and less time to amortize its cost, and thus reduces its speedup potential. Two other system programs were also used: one that compiles all polybench programs using `gcc` with full optimization (`-O3`) and link time optimization (`-flto`) enabled, and another one that archives and compresses the `/usr` directory. This constitutes the benchmark we will refer to in the rest of this section.

It is important to note that not all programs of the benchmark are used in all cases. In particular, the Dune based solution only uses the polybench programs, and runs them bare bone, *i.e.* without the support of an operating system. The Linux kernel module based

Benchmark	Program names	Remarks	bugs on Arm target
Computational - "Long"	2mm/3mm/correlation/covariance/ gemm/syrk/syr2k/symm	Subset of polybench [PY16]	correlation/symm
Computational - "Medium"	adi/fdtd-apml/floyd-warshall/ gramschmidt/lu/ludcmp/trmm	Subset of polybench [PY16]	gramschmidt
Computational - "Short"	atax/bicg/cholesky/durbin/dynprog/ fdtd-2d/gemver/gesummv/jacobi-*/mvt/ reg-detect/seidel-2d/trisolv	Subset of polybench [PY16]	cholesky/durbin
System - "Compilation"	compile	Compiles polybench code with -O3 -flto	NONE
System - "Compression"	tarbz and targz	Compresses archived /usr directory	NONE

Table 6.1: Benchmark descriptions

solution on the other hand uses all the programs of the benchmark (but one in one of its implementations), and the programs are executed under Linux in the target. Each of those restrictions and the exact programs of the benchmark actually used will be recalled in the "protocol" sections of the corresponding solutions.

6.2 Breakdown of QEMU execution time

In this section we analyze in which part of its code, nowadays, QEMU mostly spends its time during simulation. We will in particular expose bottlenecks in the simulator and relate those to our work. Finally, we detail the efficiency of the existing memory acceleration mechanisms, and show at the same time their importance.

6.2.1 Ratio of time spent in subparts of QEMU

Let us first make an approximate view of where time is spent in QEMU during simulation. To do so, we use the polybench benchmark[PY16] and two programs that build compressed archives of a compiler toolchain (`tar` followed by `gzip` and `tar` followed by `bzip2`) to represent a typical system load. It is important to note that the ratios we present are approximate due to the difficulty to use performance analysis tools (such as `oprofile` or `perf`[DM10]) on QEMU and obtain coherent results. Because of that, it can be seen on the figure that some "percentages", which in our case have been obtained with `perf`, exceed 100%. As our goal is to perform a coarse grain profiling, and since these results are overall consistent with previously published research [XTMA15], we decided not to devote more time to get a very accurate profile.

We break the simulation time down into five parts, among which four are the ones with the largest execution time on our benchmark, and the other one contains everything else. The first one is the time spent handling memory accesses outside the code cache, we define it as the time spent in `softmmu_helper` helpers. The second is the time spent executing generated code, inside QEMU code cache. The third is the time spent dealing with floating-point instructions, which are also helpers in QEMU. And finally, the time spent looking-up for the next translation block in the code cache. These can be seen in Figure 6.1. The rest of the time, that we do not present in detail here, appears in the figure with the name "others". It is mostly composed of `perf` monitors themselves taking a part of the time, plus other smaller parts of QEMU and the libraries it uses (thread locking, syscalls and various helper functions).

On this Figure we can see the weight, in percent of each of the 4 subparts we just presented. The first observation we can make is that for most programs, QEMU spends more than 30% of simulation time executing generated code. However, we can also see that it represents less

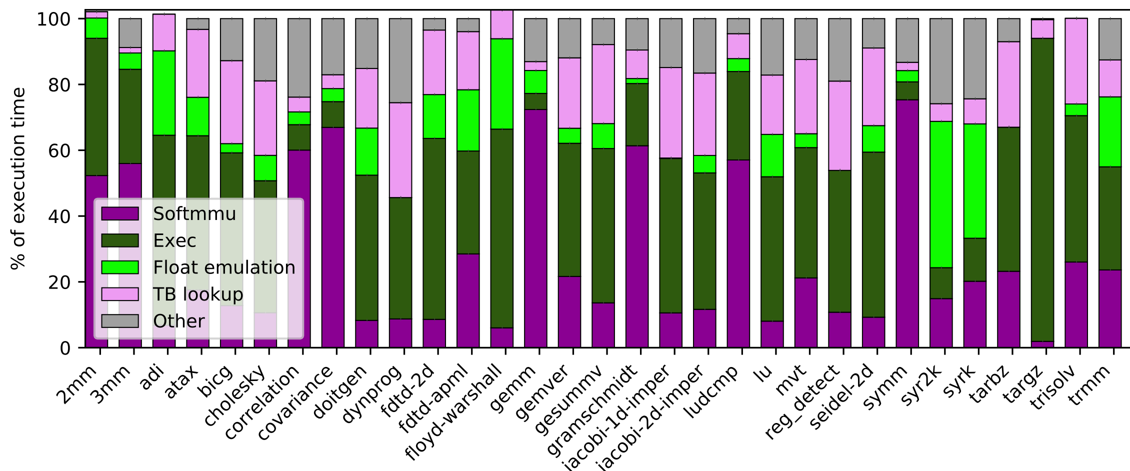


Figure 6.1: Approximative breakdown of time spent in QEMU subparts during simulation

than 50% of simulation time for almost every program. In most programs, the rest of this time is spent either in the `softmmu_helper`, in which case this time is usually represents an important part of the simulation time (> 50%). Otherwise, the time is mostly shared between lookups and fpu simulation. The translation blocks lookup usually takes an important proportion of the simulation time only for fairly short programs. In those case, we can deduce that most of the time is spent executing relatively cold code and thus no direct links between blocks have been established, forcing the simulator to go back to its top-level. The time spent in fpu computations matches what is expected, as most of those programs are working on sets of data which are large arrays of type `double`. The programs which are the longest to execute either need multiplications and divisions to perform their computations, or simply execute many operations on the data sets, such as `syr2k` or `adi`.

Overall what we can deduce from these numbers is two fold:

1. We see that the softmmu simulation contributes heavily to the execution time of programs that take a long time to simulate. This means that for those, many misses in the Virtual TLB fast path will tend to happen, and it will tend to become an important bottleneck. On mid long-running programs, this time is still far from negligible, but the time spent executing cached code and looking up for the next translation block tends to take a larger part of the simulation time. On shorter benchmarks, this tend to be even more visible.

   From this observation, and knowing that all those programs actually perform many memory accesses, plus knowing that tackling memory problems tend to provide an important speedup, we can make the hypothesis that two main profile of memory bottleneck will appear. One will be due to sheer failure of the Virtual TLB, in which case many misses happen, forcing the simulator to often call `softmmu_helper`. The other will be due to the overhead of the Virtual TLB fast path itself. Indeed, as we have seen before, while faster than a helper call, it still is not as fast as a simple memory access.

2. The other bottleneck, on the programs which make an important use of floating-point instructions, is the handling of the floating-point instructions themselves. This is not a subject we study here, the reason being that simulating floating-point operation

without using helpers, which would be an obvious optimization, is actually very hard. Each target processor having its own way to process them, even if they respect the IEEE standard as such, and according to [SBP16], clamping (the fact of using a piece of C code to simulate instructions which do not behave the same in target and host) is too often necessary to consider another solution.

To summarize, this first experiment comfort us in continuing to focus on the acceleration of memory accesses simulation.

6.2.2 Instruction breakdown: percentage of memory accesses in the benchmark

We will now interest ourselves, for each program in our benchmark set, to the percentage of memory accesses out of every executed instruction during simulation. These percentages are presented in Figure 6.2 for the case where benchmarks are running on top of a Linux target, and in Figure 6.3 when those are running bare bone. In both cases, the programs of the benchmark were run on an ARMV7 target.

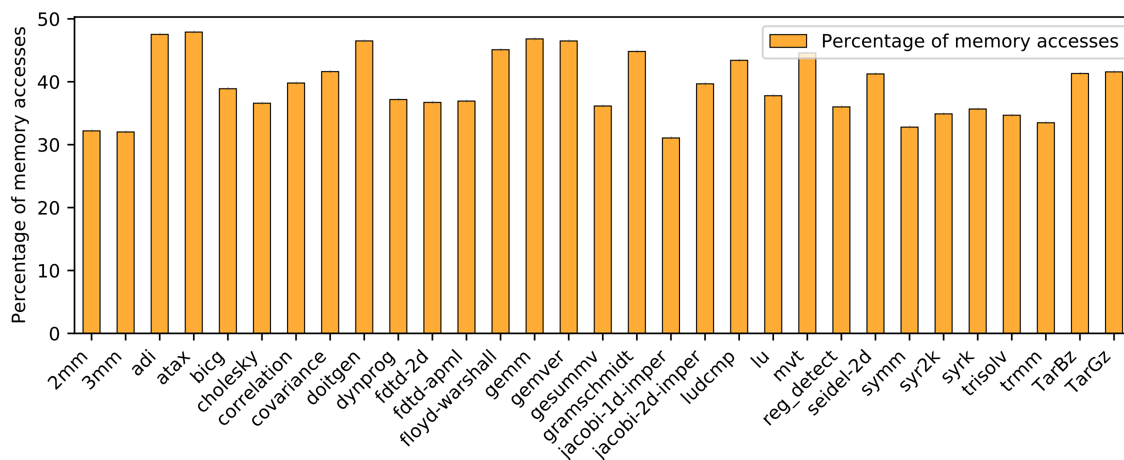


Figure 6.2: Percentage of memory accesses in each benchmark (with Linux)

A first observation to be made is that, in both configurations, the percentage of memory accesses differs. With the benchmark set running on top of Linux, this percentage tends to be higher. On the data itself, one can notice that this percentage is fairly high for every program when running with Linux. All the programs have at least 30% of their instructions that are memory accesses, with peaks up to almost 50%. In other words, between 1 out of 3 and 1 every other instruction are memory accesses in this benchmark set. This is a strong indication, once again, of the importance of the simulation efficiency of the memory accesses.

If we couple these numbers with the previous breakdown of QEMU execution, we can also make the hypothesis that in a fair number of cases, the time spent executing generated code is actually spent executing the Virtual TLB fast path. Indeed, with at least 30% of memory accesses, the performance impact of this fast path is bound to be important. The size in the code cache should also be far from negligible.

Now for the bare bone benchmark set, the percentage of memory accesses is sensibly lower. It does remain near or above 15% for most programs of the benchmark set, but it is clearly lower than when running atop Linux. What this mostly shows us is that it is important

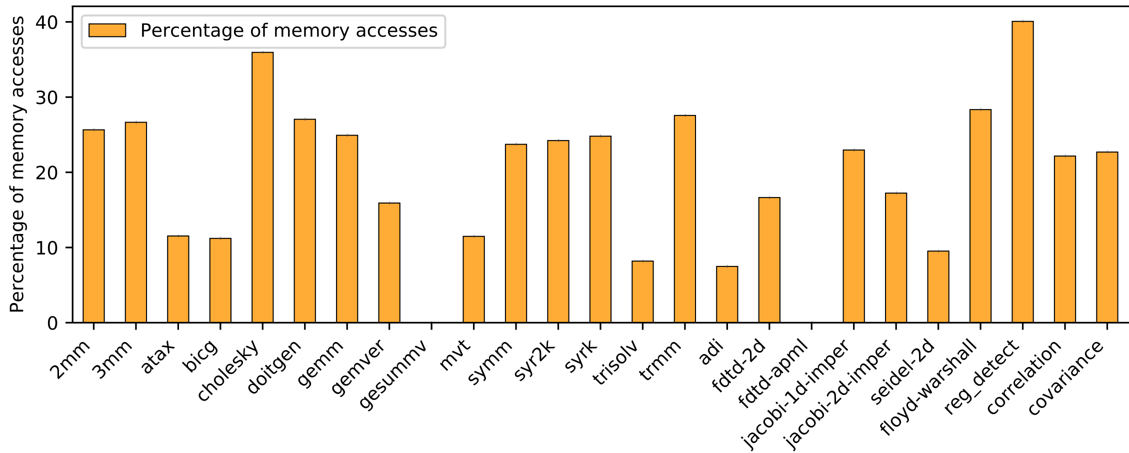


Figure 6.3: Percentage of memory accesses in executed instructions (bare bones)

to well profile our benchmark before commenting tests. The HAV based solution in particular uses the bare bone version of the benchmark, due to not being able to boot Linux, and as such direct performance comparisons with the Linux kernel module are to be taken with a grain of salt. More generally, this is also why we avoid direct comparison of other works from the state of the art, as we often do not have the exact details of their benchmark behavior.

6.2.3 Efficiency of existing memory accesses simulation acceleration

After having seen both the breakdown of QEMU performances, and the percentage of memory accesses in executed target instructions, we now detail how existing memory accesses simulation acceleration solutions behave. In particular, we study the efficiency of the Virtual TLB and the victim TLB in QEMU. To do so, we run the same benchmark with 1) no Virtual TLB fast path in the generated code and no victim TLB, 2) Virtual TLB only, 3) victim only 4) both Virtual TLB fast path and victim TLB enabled. We then provide the performances of 1), 2) and 3) relative to that of 4), *i.e.* the normal QEMU strategy.

This can be seen in Figure 6.4. This figure shows a slowdown if the y-axis value is less than one or a speedup if it is more than one. If we interest ourselves in QEMU without its victim TLB first, what we can see is that, on a few rare cases, such as `targz` or `correlation`, the simulation is actually faster than baseline. Actually, this is perfectly logical. The victim TLB lookup happens when the Virtual TLB lookup has failed, just before making a target page walk. If a program exposes a behavior where both the Virtual TLB and victim TLB often do not contain requested translations, then the second lookup will only slowdown the overall simulation. On almost every case however, removing the victim TLB results in a slowdown between 10% and 20% compared to baseline. On some programs such as `trmm` this slowdown can reach almost 70%. This does mean that the introduction of the victim TLB is already an important optimization to memory simulation in DBT.

Now if we look at the execution of QEMU with neither its victim TLB nor its Virtual TLB fast path, we can see that the performance of every program is severely reduced. Most programs suffer at least a 60% slowdown, with up to 80% for `adi`, `trmm` and others. This truly shows the importance of the fast path in the performance of QEMU. Without it, a call to a helper occurs every time a memory access is made, which costs a lot in terms of performance.

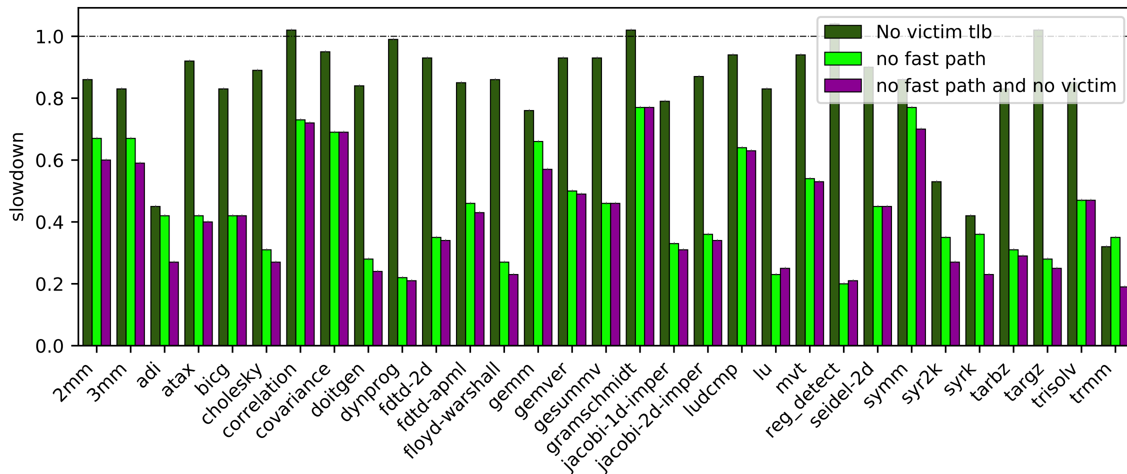


Figure 6.4: Performance of QEMU without its acceleration mechanisms

One important note to be made is that if the Virtual TLB fast path is not present in the generated code, it is still in use inside the handlers. An attempt at deactivating it has been made too, but the result is a slowdown so massive that our benchmark could not run in any reasonable amount of time, thus already giving us the conclusion. Without Virtual TLB, each memory access requires a page walk, which makes simulation completely impractical due to its slowness.

Finally, with the Virtual TLB fast path disabled, but victim TLB enabled, we see that the performance is not much better than with no acceleration at all. This is mostly due to the fact that without fast path, we already pay the price of calling the helper, which is far from free. Then, even if sometimes the victim TLB avoids a target page walk, this is not enough to make an important difference. This gives us the insight that to obtain better performances, we must fully avoid entering the helper, otherwise almost no difference will be made.

In the end, these experiments show that the existing memory acceleration mechanisms are one of the most important optimizations in DBT. From there, we can also deduce that focusing our efforts on improving memory access optimization, if we find a way to do so, may probably lead to larger speedups.

6.3 Dune based solution

In this section, we will first study the performance of the Dune based solution we detailed in chapter 4. In particular, we try to first conclude on whether the performances are better with our solution than without it. And then, to determine if we have used the whole speedup potential, we compare the percentage of speedup we obtain versus the time spent simulating memory accesses in baseline. To further show the advantage of this solution, we give insight on the communications cost between the guest, that is QEMU plus its target, and the host, which is Linux.

6.3.1 Protocol

We use the polybench [PY16] benchmark as the set of programs used in our experiments, as described in 6.1. It consists of a large variety of mathematical kernels that exhibit different

types of memory behaviors, which is crucial since the performance speedup of our solution will mainly be related to the number and types of memory accesses made by target software. The 26 programs of the benchmark were run 10 times on both baseline QEMU and our modified QEMU, all the numbers given below are the average of these 10 executions. The machine we used for running the tests, whose configuration can be found Table 6.2, was dedicated to the experiments, leading to a very narrow standard deviation.

Table 6.2: Test configuration

CPU	Intel	Xeon E3-1240 V2
RAM	16GB	DDR3, 1600Mhz
OS	Debian	Wheezy
QEMU	Git	Feb 16, 2017

All the tests are run "bare-bones", *i.e.* without an OS, on top of the integratorcp virtual board from QEMU.

6.3.2 Performance Analysis

We now detail the overall performance of our solution compared to baseline QEMU.

Figure 6.5 shows the performance speedup of our modified QEMU against the baseline QEMU. We also plot the performances of QEMU with its fast-path deactivated, so as to quantify the gain it provides.

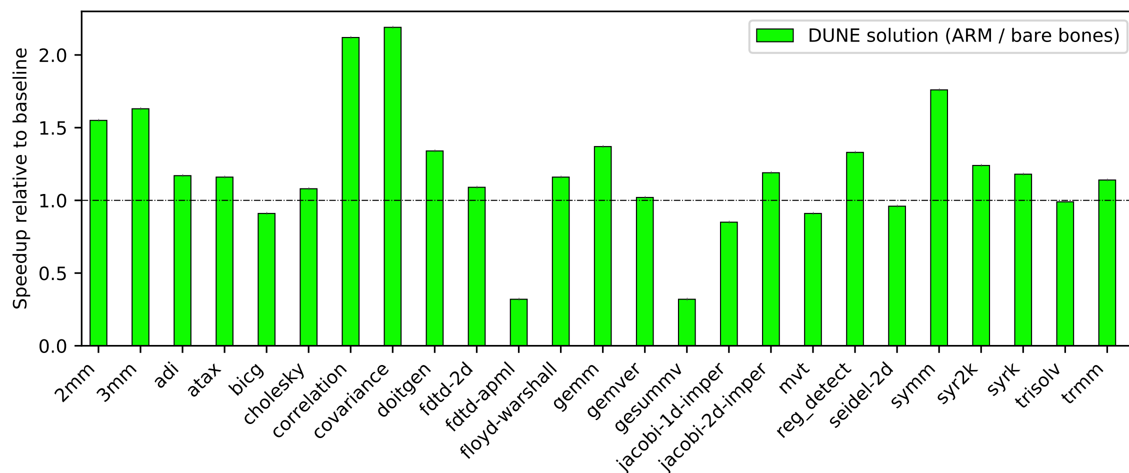


Figure 6.5: QEMU runtime with and without our solution

Compared to baseline, our solution allows for speedups up to 40%. This is the case for `2mm`, `3mm` and `symm` for example. On average, the speedup is around 30%. However, as can be seen on the figure, there are also few very short programs in which the baseline QEMU is faster. While this may seem worrying at first, it is only due to startup time, the culprit being the `dune_enter` function. This function indeed copies the whole address space of the QEMU process to the Dune environment, which is a fairly time-consuming operation.

A more interesting point to look at is the variation in speedups. As a matter of fact, the speedup per memory access is essentially the same but the simulation of a target is obviously

not only bound by memory accesses. Furthermore, performances with this solution will tend to increase with time as it will expose more and more the advantage of having a much larger memory to contain the translations than that of the Virtual TLB. Basically, at first a setup phase where the embedded translation memory needs to be filled constantly, causing page faults, happens. As such, on shorter programs for which this setup time represents a larger part of the overall execution time, the speedup will be lower. On larger benchmarks where this setup time can almost be neglected, the speedup will be larger. This trend is already visible with `2mm` and `3mm` compared to the other programs.

Now, if we look back at Figure 6.3, which contains the percentage of target memory accesses executed for each programs running bare bones, we can make a correlation with observed performances. As expected, the higher the percentage, the higher the performance gain of our solution. In particular, `2mm`, `3mm`, `doitgen` and `gemm`, which expose the highest number of memory accesses, are also the ones that expose the highest speedups. The larger this proportion is, the larger the speedups. The only exceptions being programs such as `gemsumv` which are very short and have few memory accesses overall, which mean that other part of the simulation, or even just DuneOS boot time will dominate. However, for most of the programs, higher percentages of memory accesses does translate to higher speedups.

This percentage also exposes another important information. No matter how good our optimization is, it only concerns memory accesses. As such, if 90% of the instructions are not memory accesses, the speedup we can obtain by infinitely accelerating the memory accesses is $\frac{1}{90} \approx 11\%$. The achievable performance of this solution is then dependent from the fact that most programs perform a fairly high number of memory accesses, with up to one every other instruction on a RISC CPU, according to [JRHC95].

To further explain the speedups, we traced the number of calls to the *slow path* in baseline QEMU versus our page fault handler in our modified QEMU. The result can be seen Figure 6.6, with the *slow path* in green and the page fault handler in purple. First, it seems quite striking that the number of calls to the handler in the case of our solution is much lower than that of the *slow path* in baseline QEMU. For example, in the `2mm` benchmark, more than 2 billions "translation misses" happen in the case of baseline QEMU, while our modified QEMU only experienced around 10 thousands page faults. For most benchmarks, the percentage of misses is in the order of 10^{-6}, which could be considered to be pretty much negligible. This means that the overhead of memory accesses has been dramatically reduced.

But this raises the following question: why is our modified QEMU not capable of higher speedups? A first hint has already been given by [XTMA15] which explains that memory access management amounts to 40% of the simulation time Another hint can be found directly in Figure 6.3 where we can see that in the benchmark set we used for these tests, usually between 20 and 30% of executed instructions are memory accesses. Consequently, as Amdahl's law[Amd67] teaches us, by accelerating from a factor s a fraction F of the execution, one can not expect a larger performance gain than $g = \frac{1}{1-F+\frac{F}{s}} \approx \frac{1}{1-F}$ if $s \gg 1$. A second element of response can be extracted from Figure 6.5. There, the execution times of QEMU with its *fast path* disabled have also been measured. What one can see is that going through the *slow path* does not induce extremely high overhead (though it does almost double execution time). In other words, there are definitely other bottlenecks in QEMU. Some such as vector ISA have been studied in other works such as [MFP11], and it was indeed shown in our previous section that the execution time spent simulating vector ISA can be fairly large. This means that avoiding that path will not always grant the 66% performance improvement (the highest percentage of time spent execution `softmmu_helper`) one could hope for, as sometimes other parts of the simulator will dominate the overall execution time.

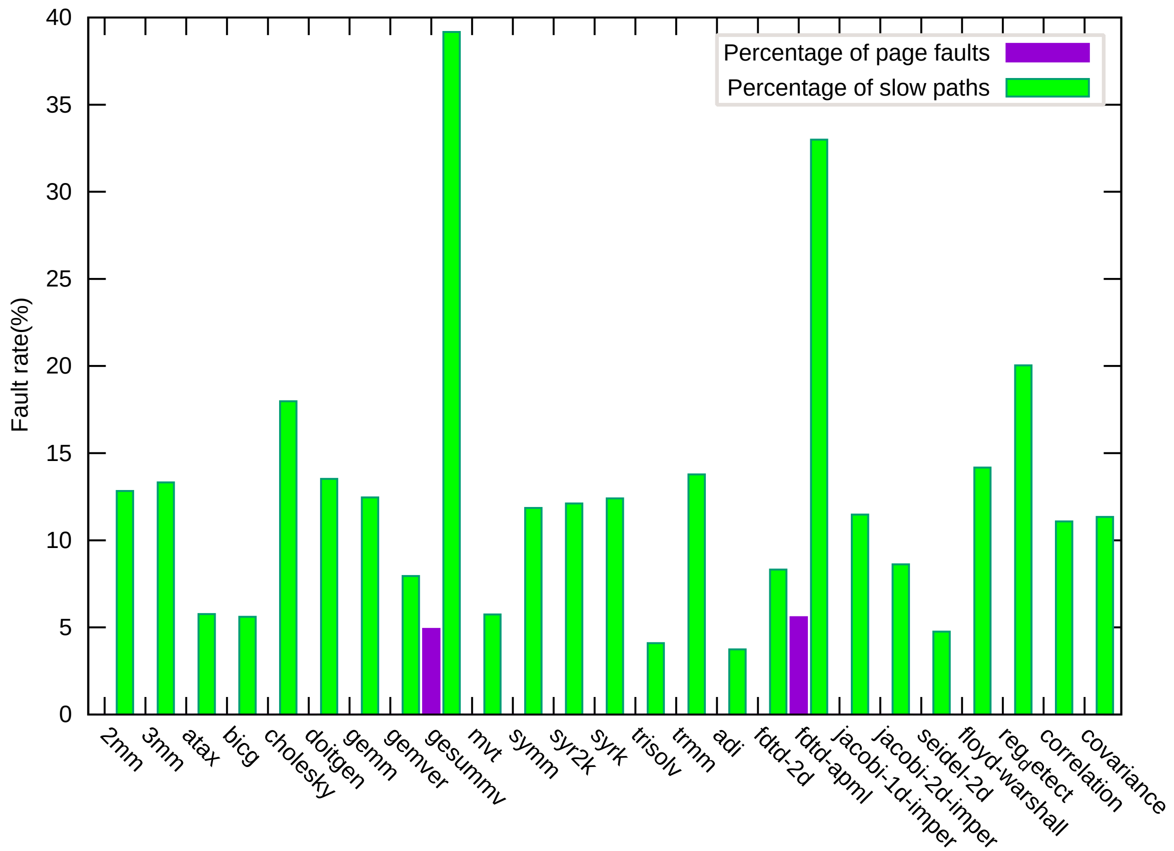


Figure 6.6: Percentage of calls to *slow path* or page fault handler

However, this figure does show that our insight about the possibility of reducing the number of accesses to the *slow path* was correct. And it does show that reducing this number will bring better performances.

However, it needs to be noted that in this work, which can only run bare bone programs, the number of memory accesses is actually fairly lower than normal. This was confirmed by looking at figures Figure 6.2 and Figure 6.3. There one can indeed see that the number of memory accesses when running the same benchmark on top of Linux is clearly higher than bare bone. This being mostly due to the presence of context switches and the execution of other processes, as well as the OS itself.

6.3.3 Communication costs

A cause of worry we could have concerning this solution would be the cost of communications between QEMU (the guest here) and the host (Linux). Indeed, as we said in the state of the art, work from [SFGP15] shows that this cost can sometime be prohibitive. However, the whole simulator is running on top of the virtual CPU for us, as such communications with the host will be fairly limited. Basically, in our case, communications almost only happen for syscalls. According to the micro benchmark provided by Dune, the overhead on each syscalls is around one thousand cycles. We then count the number of syscalls in QEMU for an

average run to have an idea of the overhead caused by these communications, these numbers are shown in Figure 6.7.

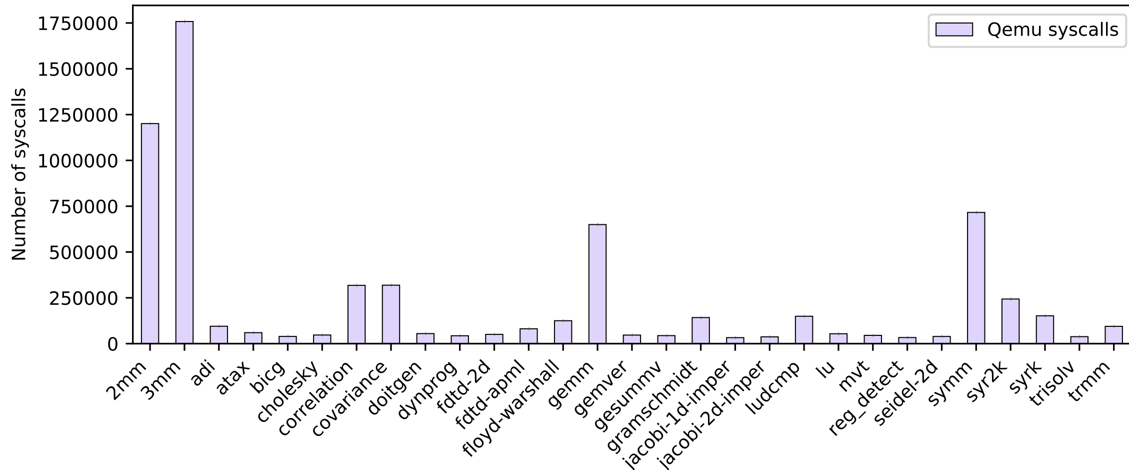


Figure 6.7: Number of syscalls during simulation

We can see on this figure that the highest number of syscalls in a simulation is around one million seven hundred fifty thousand. While this number may seem high, this means that at worst, considering the syscall overhead we computed for Dune, it represents less than two seconds in the whole simulation. Since in this case the simulation runs for several minutes, this means that the final overhead will be pretty much negligible. Furthermore, this does not count that, actually, Dune is able to process some syscalls directly in HAV context.

6.4 Linux Kernel Module solution

The evaluation is done running a Linux target OS on an Arm-based target ISA and an i386-based target ISA. The host is an Intel x64. The Linux target runs in an *initramfs* to focus the evaluation on the simulation of memory accesses and not the emulation of disk operations. The configuration details of the host machine are given in Table 6.3.

CPU	Intel Xeon E3-1240V2 @3.4GHz
RAM	16G @ 1600Mhz
OS	Archlinux (kernel 4.14.15)
QEMU version	git clone as of January 24th, 2018

Table 6.3: Setup Details

All benchmarks from 6.1 are run here, on a “virt” QEMU board with a linux target.

The measures concerning percentage of execution time and cache metrics were acquired with Linux’s `perf` tool [DM10]. To have meaningful results concerning the execution times *per se*, we measure them using wall clock time. This is done using Linux `time` command in the target, as by default QEMU virtual time is kept aligned with host time, so reading the target cycle count register (`ccnt` on Arm) returns the value of the host cycle count register (`tsc`, the time stamp counter on x64). Thus, `time` returns a measure of time with a precision of one hundredth of a second.

All metrics concerning numbers of calls to the slow path were extracted as follows. The number of calls was measured across a full run *i.e.* a Linux boot plus a program run. Then, we subtracted from this number the average number of calls to the slow path during a Linux boot. This gives us a close approximation of the number of calls to the slow path during the program itself.

Our prototype currently handles Arm-based and Intel-based target, both only 32-bit. Also, only an Intel x64 host was tested, however, using other hosts should not cause any major issue, as both the Linux Kernel Module and QEMU backend do not make use of any Intel exclusive features.

Finally, throughout the experiments, we use 3 simulators: baseline QEMU, our "plain" solution in which all accesses are handled through the mapping approach described Section 5.2, and our "hybrid" solution, described at the end of same Section, in which the kernel code still goes through QEMU `softmmu_helper` and Virtual TLB code path.

6.4.1 Performance Overview

Let us now examine the performance of the hybrid solution compared to baseline QEMU. The raw runtimes of both solutions are presented in Figure 6.8. What can first be seen is that,

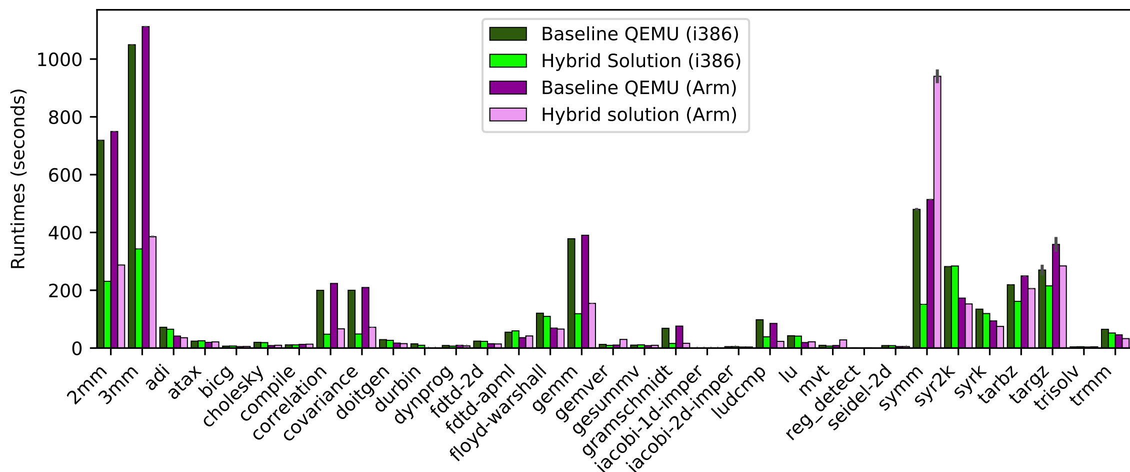


Figure 6.8: Programs runtimes

on the "Long" programs especially, our solution tends to perform better than baseline. On "Medium" programs the speedup is slightly reduced. And for the "Short" ones, runtimes are often almost the same as baseline or slightly below. The overall Arm target runtimes are also higher than i386's ones. The speedups and slowdowns can be seen more precisely on Figure 6.9. The first thing to notice here is that, over the 33 programs, 11 show a slowdown for Arm and 5 for i386 compared to baseline. The second is that the speedups can reach up to four times baseline performances, and several programs expose speedups above three, leading to an overall clear performance gain over the 33 benchmarks. Looking more closely, we can see that a vast majority of the slower programs belong to the "Short" benchmark (`atax`, `bigc`, `gemver`, ...). In these cases, our approach pays the cost of installing the translations while not reusing them much. The special case of `symm`, for which we have a large speedup on i386 and a large slowdown on Arm, will be analyzed later.

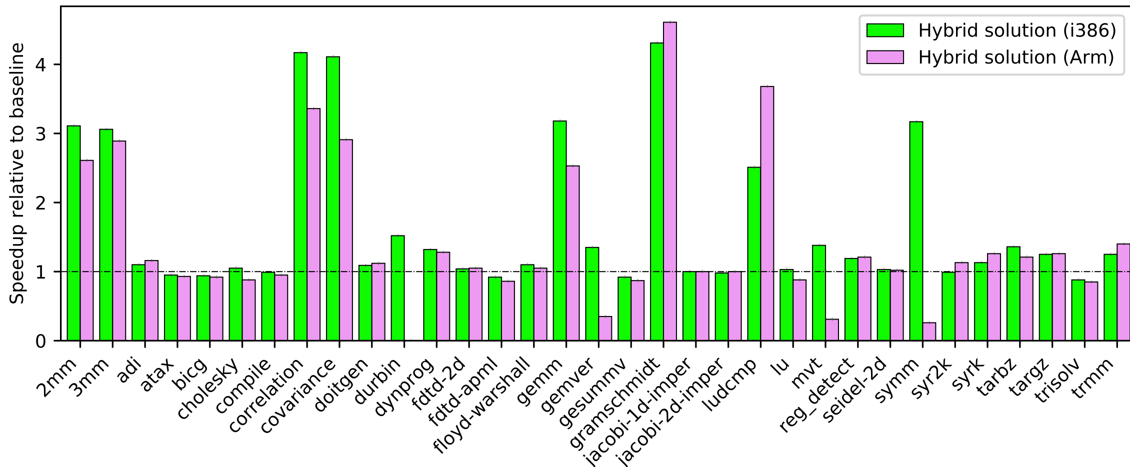


Figure 6.9: Programs speedups

6.4.2 Hybrid solution analysis

Let us now examine more specifically the gains brought by using the hybrid solution instead of simulating both user and system code through our aliased target address space solution (noted "Plain solution" in the figures). A comparison of performances, given as the speedup

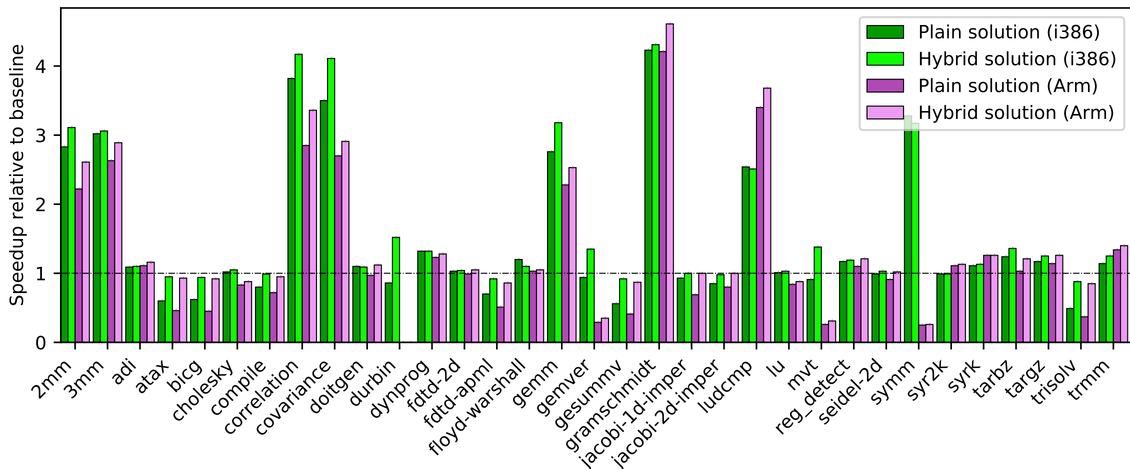


Figure 6.10: Speedups relative to baseline with both our plain and hybrid solutions

compared to baseline QEMU using the mapping for both user and system code or only for user code, can be seen in Figure 6.10. Every benchmarks above the dashed line represents a speedup, those below a slowdown. What can be seen is that, while multiple benchmarks expose some speedup in case of our plain solution, there are also quite a few which have important slowdowns. The highest speedup is already three, but the highest slowdown is also around three, which means that overall the payoff is much more mixed. This shows that using a hybrid solution does indeed bring higher performances than a fully aliased mapping based solution.

Figure 6.11 characterizes the problem when using our plain solution. It shows the ratio

of time spent simulating target system code. We can see there that for slower benchmarks, the ratio of time spent in target system code is fairly high. While executing `durbin` on an i386 target for example, 31% of the time is spent simulating target system code. For that same benchmark, the slowdown of the plain solution compared to baseline is around 14%. In other words, there is a high correlation between time spent simulating target system code and performances with this solution. In comparison, the same benchmark with the hybrid solution exposes a speedup of 52% compared to baseline QEMU. Another interesting point is Linux boot time, which takes about the same time with our hybrid solution and baseline, but exposes important slowdown with our plain solution. Linux boot is actually a typical pure system workload, as such, its respective performances on both version of the solution clearly exposes the problematic behaviour of our plain solution on system code. Since the goal of our work is performance optimization, it is worth to only apply our solution wherever it is truly useful, and where it does not cause negative impact, *i.e.* on user code.

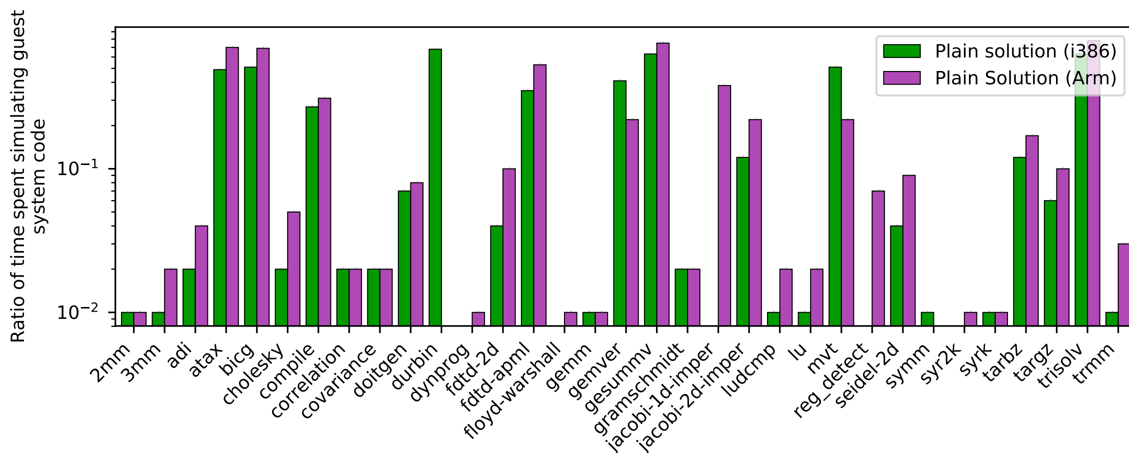


Figure 6.11: Ratio of time spent simulating system code

6.4.3 Address space mapping analysis

Let us now analyze more precisely the performance of the target address space mapping, and in particular the speedups factors.

The first and foremost speedup factor, that follows directly from the use of the aliased target address space mapping, is the native handling of target memory accesses. Basically, compared to the software solution, target memory accesses are now handled through much fewer operations, which reduces their overhead. In particular, if we look back at the code in Figure 5.6, we suppress the instructions that cost the most in the original code, since there are no more branches, and that there is only one memory access.

Table 6.4: Instruction-cache misses in both baseline and hybrid solution

benchmarks:	long	medium	short	compile	compression
Baseline QEMU	2.16×10^9	1.23×10^9	1.13×10^9	2.35×10^9	2.00×10^9
Hybrid Solution	1.52×10^9	1.26×10^9	1.17×10^9	2.24×10^9	2.05×10^9

A second speedup factor, which is the consequence of the previous one, is instruction cache locality. A measure of the number of instruction cache misses for the three types of benchmarks is given Table 6.4. We can see there some gain in computational programs, which incidentally expose the highest speedups, where the instruction cache miss rate does indeed decrease. However, for more system oriented benchmarks such as compilation and compression, this gain is fairly shallow. This is mostly due to the effect of traps. Whenever an instruction traps, we have to execute the kernel page faults handler. This code is fairly large, and as such, the instruction cache will quite probably be filled with an important part of kernel code. Therefore, when we return to target code, misses are bound to happen. This however does not invalidate the benefits of our solution. We still obtain an important speedup, as exposed before.

More generally, the trap mechanism greatly impedes performance. Whenever the execution of the slow path is necessary, instead of having "just" to call a helper that implements it, our solution leads to the execution of a full trap and its associated handlers. While the time spent handling a trap in Linux is not easy to measure, it can easily be assumed to reach thousands of cycles. As such, slow paths are usually much slower using our solution than with baseline QEMU. This means that MMIOs, target faults, as well as accesses after mapping flushes, will slow down the overall emulation. Fortunately, our solution has another highly

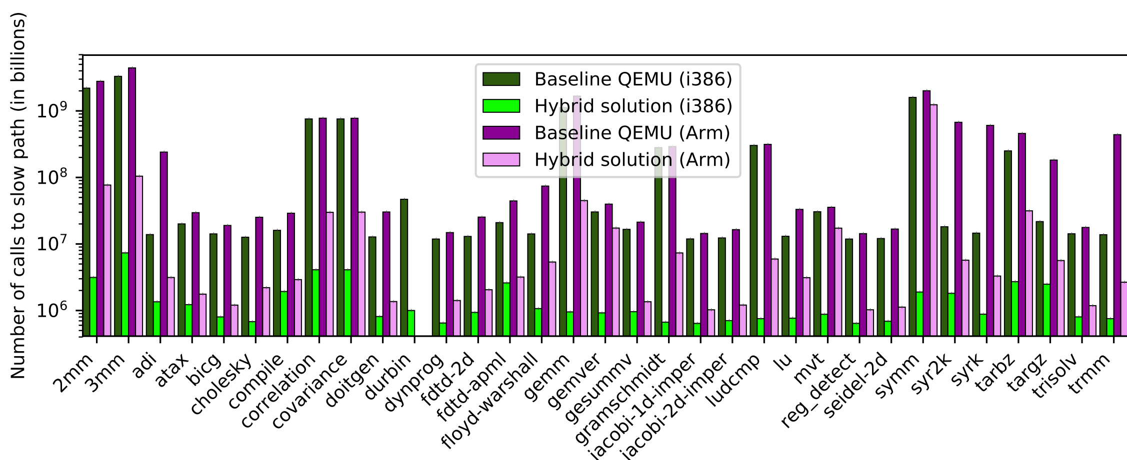


Figure 6.12: Number of calls to slow path during program execution

beneficial advantage. Compared to QEMU Virtual TLB, it can potentially hold the whole target mapping at any time. This yields the results visible Figure 6.12, in which the number of calls to slow path can be seen for both our solution and baseline QEMU. The first thing to notice is that, in most cases, the number of calls to slow path is massively lower in our solution than in baseline. The order of magnitude for the number of calls to slow path with baseline solution is in billions, while for our solution, the number of traps is in the order of millions. This difference easily allows to outweigh the loss in performance due to a single execution of the slow path.

This does set the main key to estimate the performance of this solution. Basically, what we need is to have time to amortize the cost of the costlier slow paths.

This explains the fact that very short programs, such as `trisolv`, even though they are computational, do not expose any performance gain, or even expose slowdowns. The principle is that, if either the benchmarks are too short for the fast path performance gain

to be seen, or if too many flushes happen (many context switches for example), then the performance gain of our solution will be close to zero, if any. This is further reinforced by the fact that, in those cases, the number of calls to slow path in both baseline and our solution is fairly similar.

We also need to explain why `gemver`, `mvt` and `symm` have an important slowdown on Arm. These programs indeed exhibit a very special case for our simulator. They go through large arrays column per column instead of line per line. The result is that, during a first run, every access will go through a trap, leading thus to a large setup time. This is clearly also the case on Intel target, but there `symm` is amongst the highest speedups. However, on Arm, Linux causes preemption much more often than on Intel, thus causing costly full mapping flushes. For `gemver`, this goes from around 70 context switches on Intel to around 320 on Arm (measurements were made monitoring `/proc/pid/status` and are not fully accurate). Brought back to a per second measurement, this is around 50% more on Arm. Also, QEMU `softmmu_helper` code that performs the page table walk is slower for Arm than for Intel. On Intel, a call to `softmmu_helper` is between 100 and 500 ns (most of the time below 300 ns), whereas on Arm it is between 1 µs and 3 µs. As such, the result is that, on Arm, this setup time will happen much more often and be slower in itself, thus causing an important slowdown. From measurement, the first full walk of a 4096 entries array on Arm takes between 10 ms and 20 ms with some rare spikes at 1s, whereas on Intel it almost never reaches 1 ms, and stays around 300 µs most of the time. This is a perfect example of a case for which the cost of internal page faults is not amortized.

Table 6.5: Percentage of time spend handling the target MMU with our hybrid solution

benchmark:	long	medium	short	compile	compression
Hybrid Solution	5.31 %	6.85 %	37.15 %	36.39 %	3.27 %

As an insight of what is left to be further enhanced, Table 6.5 presents the percentage of times spent, with our solution, handling the target MMU. The same metrics for baseline can be seen in Figure 6.1. In the case of our solution, this percentage counts both the time spent in our handlers (for user code) and in the `softmmu_helper` (system code). What can be seen first is that this acts in line with what we expect in term of speedup for each group. On the “Long” group and compression, the ratio of time spent handling the target MMU is much lower with our solution. On the “Medium” group, this ratio is slightly higher in our case. However, the overall runtimes remain better in our case, but with lower speedups. Finally, for the “Short” group and `compile`, the ratio of time spent handling target MMU is much higher in our case. Overall, our solution provides no speedup for this group, and we even have some slowdowns.

It is to be recalled though, that memory accesses simulation do not resume to target MMU management, but also to the execution of the memory accesses themselves. This time however is fairly hard to measure in QEMU. On the other hand, we can see in Figure 6.2 the percentage of memory accesses out of the total number of executed instructions. This numbers complement our previous analysis, as even on the medium group where not much gain is made concerning target MMU management, we still gain on the 30% or more of instructions which are memory accesses. On the short group, these 30% are no longer enough, and the time lost in internal page faults can no longer always be amortized.

6.4.4 Performances with in kernel fault handling

Studying the group where we have no speedup, we see that the main cause is the relative slowness of (internal) page fault handling of our solution. In particular, if the benchmark we run causes a very large proportion of internal page faults, *e.g* faults that are only used to update our mapping, this cost will be extremely visible. On the general case we could think this remains mostly unseen, but as a matter of fact, even on "good" programs, we can still extract an important speedup by accelerating internal fault handling. This is why we provide the solution detailed in Section 5.4, in which we begin to manage internal page faults directly in kernel. We call this solution "proof of concept", as its implementation only handles page faults due to read accesses in the kernel, the ones due to the writes being still forwarded to our handler in QEMU. The speedups provided by this method and metrics explaining those

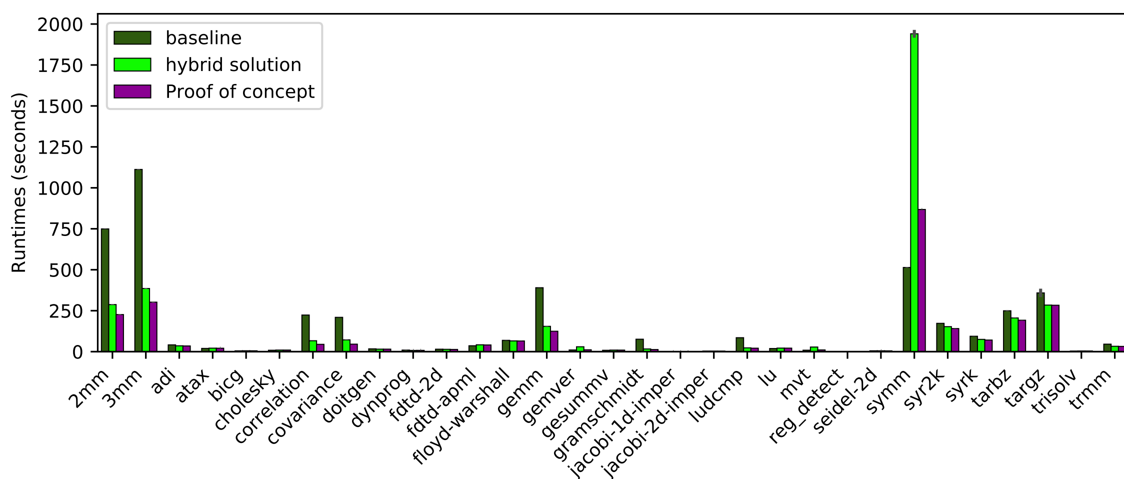


Figure 6.13: Runtimes with our proof of concept, hybrid solution and baseline

will be detailed in this section. Throughout this section, we consider that the target is a "virt" board from QEMU with an ARMV7 CPU, running on an Intel x64 host.

First, let us look at Figure 6.13 where the runtimes for the hybrid solution, baseline QEMU and our proof of concept are shown. What we can mainly see is that, even on longest programs such as `2mm` and `3mm` where the speedups of the hybrid solution was already fairly high, our proof of concept still offers an important improvement to performance. We can also see that the slowest program, `symm`, is running much faster with our proof of concept than with the hybrid solution, even though it is not yet as fast as baseline.

Looking at Figure 6.14 we see the more precise performance of our proof of concept solution relative to baseline QEMU, with the hybrid solution relative performances for reference. Overall, the average speedup with the proof of concept work is of 77.81%. For reference, the average speedup with the hybrid solution is of 46.25%. And the speedup in the case of [WLW+15] is of 63% (1.63 times faster), but on a completely different set of benchmark on a different setup as they use Android Emulator. Most importantly, as the proof of concept work is only implemented for read accesses, there is still room for speedup improvement.

We can observe in particular our three pathological cases: `gemver`, `mvt` and `symm`. `symm` is now about 2.5 times faster than with the hybrid solution, which is still about 40% slower than baseline. This is due to the fact that it accumulates a pathological behavior for both read and write accesses. As such, we only solved about half its problem. `gemver` has about

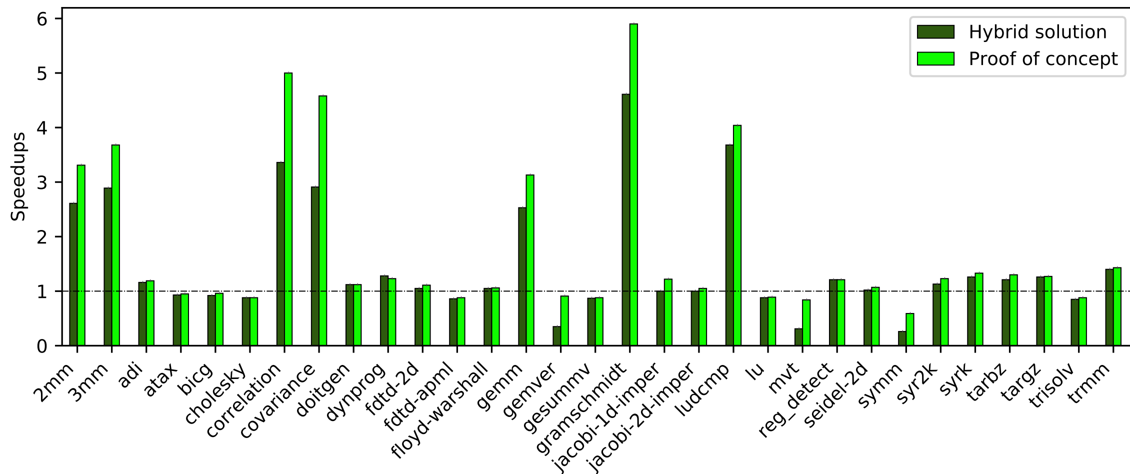


Figure 6.14: Speedups of our proof of concept compared to baseline, with hybrid solution speedups for reference

the same speedup compared to the hybrid solution, which brings it to now be only about 9% slower than baseline, `mvt` is also almost in the same case, but with around 16% percent slowdown compared to baseline still. In both of those cases, we have already got much nearer to performance parity with baseline. And an important speedup potential remains.

Even for `2mm` and `3mm` which already exposed respectively a 2.61 and 2.89 times speedup, an important improvement was made, with now 3.31 and 3.68 times speedup. `gramschmidt` for which the hybrid solution provided a near 5 times speedup is now near 6 times. Overall this does show the extreme importance of the page fault mechanism costs in the overall performance of embedded address spaces based solutions. And most importantly, even though the solution could potentially slowdown the execution of faults in the case where the simulator still needs to handle the fault, no overall slowdown was seen. In other words, here the added cost is always amortized thanks to the overall gain.

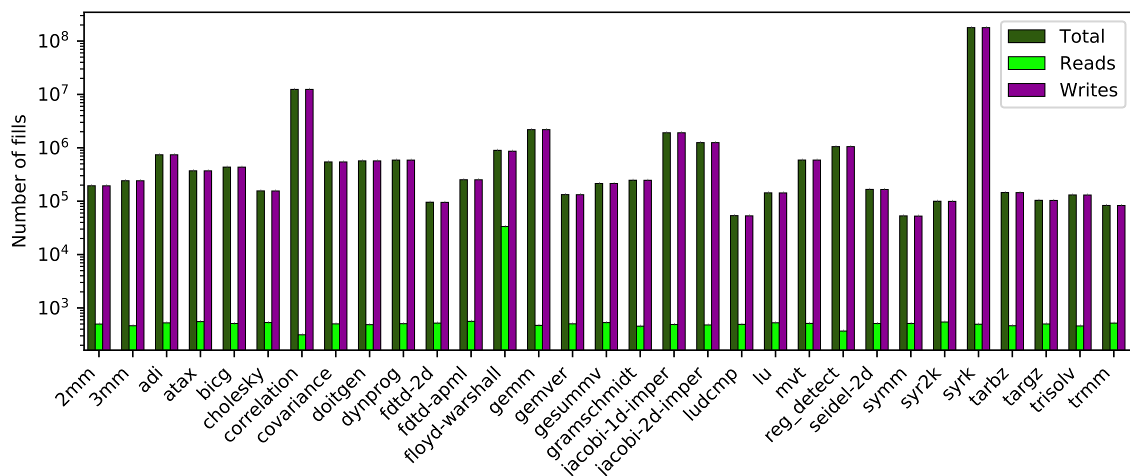


Figure 6.15: Number of calls to user side handler with our proof of concept

This overall gain can be further explained by studying the results presented in Figure 6.15. We see there the total number of times the user side handler has to be called to process the fault, the number of times it has to do so for a read-access, and the number of times for write-accesses. The progress brought by the proof of concept work on read accesses is particularly spectacular. On every program composing our benchmark, the number of user side handler calls is massively lower for reads, where our solution is implemented, than on writes where it is not. And on every program but one, this number is near negligible. Considering that any call to user side handler also implies an `ioctl` trip back into the kernel, plus the call of full QEMU softmmu handlers, we can consider that we save between 2 µs and 4 µs (1 µs for an ioctl, between 1 µs and 3 µs for a `softmmu_helper` call for an Arm target). Although it should be noted that a part of the `softmmu_helper` call, although manually inlined, has to be done in kernel, so the gain will more likely be around 1 µs and 2 µs per call, which is still far from negligible. This then explains our overall gain with this method.

These facts lead us to thinks that once we finish this prototype we can hope to reduce the number of costly fills to be near negligible overall, and we can expect even larger speedups at that time. This will also naturally extends the speedup to the pathological cases we identified but that rely on mostly write accesses, or on both read and write accesses as `symm` does.

6.5 Conclusion and comparison of both methods

In the end, we have first confirmed how important the memory accesses are regarding the performance of DBT based simulation. Indeed, with sometimes more than 60% of the time spent simulating the MMU, and with probably an important part of the instructions in the code cache being used to simulate memory accesses, we can see that the execution heavily depends on their performance. This is even more striking once we begin to remove existing optimizations. Without the victim TLB, which is the least important one, performances can sometimes be halved. Without the Virtual TLB fast path, performances massively drop on every program we tested. And with no Virtual TLB at all, the simulation is just unpractical, as even the simplest programs can take the better part of a day to run. In other words, the memory subsystem of DBT is probably one of the most important.

We have then seen the performance of the two solutions we propose to accelerate these accesses, one based on HAV, through the Dune framework, and one based on a Linux kernel module. In both cases we have seen a fairly important speedup over baseline, at least in average. The Dune solution gives us around 40% better performance on average, while the Linux kernel solution gives around 78% better performance with its proof of concept implementation.

The solution based on Dune gives better performances than baseline in every case, thanks to very low communication costs. Unfortunately, its peak speedup is lower, and as discussed in chapter 4, it is much more limited in functionality and debugability. It is however, truly the most consistent approach in term of raw performance. Its communication costs are extremely low, and communication itself is relatively straightforward to implement.

The Linux Kernel module based solution can be implemented using two methods. The first one, which forwards page faults to handler in QEMU, achieves a speedup on an Arm target of around 46%. The second one exposes in average 78% better performances than baseline, but it requires implementing part of the simulator in the kernel module. This especially stresses the importance, in those methods, of reducing communication costs to their strictest minimum. In particular here we have worked with the goal of avoiding round trips into the kernel. This also shows that going further does require an important work, and

rework, of the simulator.

In the current form of this work, the balance would go in favor of the Linux Kernel module. Especially since it still has margin for improvement. However, the Dune based solution also has improvements yet to be explored, in particular for 64 bit targets, that would not be possible to achieve as easily with the Linux kernel module.

Furthermore, if we compare our Linux Kernel Module to the most similar approach from the state of the art, HSPT [WLW+15], we do obtain better performance with our proof of concept work. The benchmark we use is not exactly the same, so we cannot make a fully direct comparison. However, on their benchmark, the average speedup they obtain is of 63%, whereas we obtain about 78% speedup with our proof of concept work. Once again, the importance of communications costs seem to be explaining our enhanced performance here.

Finally, if we compare it to a non-invasive approach such as the resizeable Virtual TLB from [HHC+16], we see that actually this approach tend to provide slightly better performance on average (89% speedup for this solution). This can probably be explained by two reasons. First, this approach only enhance the existing Virtual TLB, in other words, it does not add any communication overhead as we do. The second is that even operations such as flushing the Virtual TLB are fairly optimized, due to its size being pretty much optimal for the current running process. At the same time, it pretty much solves the problem of calls to the `softmmu_helper`. But while these observations might seem worrying about our method, they actually are not. It indeed needs to be reminded that we also reuse QEMU Virtual TLB, as our solution is actually a hybrid which offloads the management of system code to the legacy code path. As such, the improvement made by [HHC+16] to the Virtual TLB would actually be useful for us. Our solutions are actually fairly complementary, and that encourages us to implement the resizable Virtual TLB in our solution too, in future works.

Chapter 7

Conclusion

This thesis is devoted to the acceleration of memory accesses in dynamic binary translation. At the end of the problem statement, we determined that we would have to answer, during this work, three questions relative to this topic:

1. What schemes may be devised so as to accelerate target memory accesses simulation in dynamic binary translation?

2. What are the most efficient methods we can think of to implement those schemes?

3. Are the designs we choose to accelerate target memory accesses simulation implementation specific, *i.e.* tied to the underlying OS or framework we use, or can they be reused atop any of those?

To the first question, we found two answers. The first one is to reuse a technique which was developed in the context of same-ISA simulation: Hardware Assisted Virtualization. With this solution, we provide a way to use HAV in the context of cross-ISA simulation. To do so, we use our simulator as a special guest, which could then behave as an OS for the simulated target code. Thanks to that, we were able, in particular, to manipulate the simulator memory so as to provide it with a view of its address space that is as close as possible to the real hardware view. And since we act as an OS, we can leverage the host hardware to do the heavy lifting for us, thus importantly reducing the overhead of managing target memory address space.

The second answer we found was to use the host OS directly instead of relying on HAV, and we devised an OS-like scheme for the simulator. In this scheme, we use a companion kernel module for the simulator.

This companion module is tasked with the host virtual memory manipulations. It provides the facilities to create an address space for the target, embedded in the simulator address space. Just like the HAV based solution, we leverage the host hardware to perform the virtual to physical addresses translations, including for the target address space. This allows to obtain speedups of almost 78% on average, for the most advanced version of this work.

Answering the second question is pretty much a matter of comparing the two approaches we used to answer the first one. If we are first purely interested in performance, then making a clear decision now is complicated. The HAV based solution has more consistent performances, mostly due to not suffering from a communication bottleneck. But its peak speedups are also clearly inferior to that of the kernel module solution, although this might be

due to the difference in execution of the benchmarks (bare bones for HAV, atop Linux for the Kernel module). The Linux kernel module solution on the other hand is less consistent, with a few benchmarks being slower than baseline. But both its peak and average speedups are better than the HAV based solution. It also has a fair margin of progression as we described before.

On the ease of maintenance, the Linux kernel module solution is clearly superior to the HAV based solution. The Linux kernel module itself is quite lean, and most of the code is written using Linux host architecture agnostic abstractions. QEMU itself is only very slightly modified, with all of its standard code paths functioning normally. Updating the module to newer Linux versions is usually fairly simple, and most of the time only a matter of recompiling the module. On the other hand, the HAV based solution relies on Dune [BBM+12] which is far heavier (Linux kernel module, plus a LibOS) and no longer maintained. It also requires the simulator to use host specific code, in particular for page walks or TLB flushes. Worst, with Dune debugging is much more complicated, standard gdb based debugging tend to now work, and crash. It is also hard to determine whether bugs come from Dune or from the simulator. As such maintenance of the HAV based solution is clearly harder than for the Linux kernel module.

On the evolution potential of the solutions, the HAV solution does show some promise. If we can manage to replace Dune, it is highly likely that supporting efficiently 64 bit target will be easier than for the Linux kernel Module. We could basically extend the simulator 0S like behavior, and create a full page table for the target address space. This would however also require a rework of QEMU, in particular, to avoid constant context switch, it would have to change from a basic block based code generation to a trace based code generation paradigm. QEMU helpers could then be seen as syscalls, provoking the context switches to simulator address space when needed. For the Linux kernel module, most of the evolution will come from the integration between the kernel module and the simulator. This also requires a rework of QEMU so that multiple of its data structures may easily be shared with the kernel. The support of 64-bit targets will also most likely need to be done through dynamic allocation of the target address space, which would require more indirections in the fast path (this method will be explained in the prospective section below). In term of raw performance, the perspectives of the Linux kernel module solution are also fairly positive. As we have shown through our experimentations, just implementing the management of read accesses directly in kernel boosted the speedups of the solution, compared to baseline, from 44% to 78%. Once we implement the same solution for write accesses, we can expect speedups to be even higher.

Overall we tend to think that, as things are now, the Linux kernel module solution is the most reasonable to use. However, an HAV based solution, with a simulator specifically tuned for it, and as has also been shown by CAPTIVE[SWF16], can be fairly promising.

In fact, if we look at our third question, we might say the second question could have a third answer. Looking at both solutions, strictly from the design point of view, we can see that at least the simulator side is almost the same. The problems are also similar, except for the communication problem which is non-existent in HAV based solution. But essentially, we need to reduce the cost of non RAM or target faulty accesses, and we need to manage efficiently our page table. We would also need the simulator to be as aware as possible of the embedded address space to ease its implementation. And the sharing of data, such as target physical to host virtual addresses could, in both case, allow enhancing performances and ease implementation.

At the end, we can say that, for now the answer to the second question is to use a Linux

kernel module based solution. But in the long run, a well tuned, custom simulator would be a wiser choice. Its design could fairly easily be shared between an HAV and linux kernel module based solution, through some abstractions. Even today, this is actually partially the case as both methods do have a fairly similar designs, with an important part of the code being shared. Most of the existing difference in the simulator code between the HAV and Linux kernel module based solution are due to optimizations and code quality improvement. The existing code could with reasonable effort be, with some wrappers, be adapted again to the HAV based solution.

7.1 Prospective

While this thesis has already given satisfying results for an academic work, four points are left to be addressed. Those are:

1. N privilege levels support,
2. QEMU multi-threading support,
3. 64-bit targets support, both in HAV and Linux kernel module designs,
4. In kernel write accesses management with our proof of concept Linux kernel module, which essentially amounts to rethinking the simulator.

In this section we will present our roadmap to address those issues.

7.1.1 N privilege levels support

In Chapter 5 we briefly commented on the problem of supporting multiple privilege levels. Indeed, since the target memory is accessed inline in generated code, no software control is done over it. As such, if a target user level process tries to access a target supervisor page that is already mapped, the access will not trigger a target fault as it should.

In this case, we dismissed it as, for the traditional supervisor-user mode model, we solved it as a by-product of our main performance optimization, a hybrid solution that uses QEMU Virtual TLB for supervisor accesses, thus effectively avoiding the problem. Furthermore, the problem is more security related than functionality related in this case. Only a piece of target code that would voluntarily attempt to access kernel code would risk doing so, in other words, this only concerns malicious programs.

On the other hand, there is an important case which is not covered here: target virtualization capacities simulation. If the target has a hardware assisted virtualization mechanism then it implies at least two more privilege levels over the standard supervisor-user: guest-supervisor and guest-user. Obviously, if managing two privilege levels through standard means was not possible, supporting even more will not be possible either.

However, we can build up on our existing solution, and on the way QEMU Virtual TLB manages a similar problem. Indeed, the Virtual TLB also needs to avoid code from privilege level X to access memory reserved to a privilege level higher than X. To do so, the entries are split in groups of privilege levels, *e.g.* there is not a unique Virtual TLB, but one Virtual TLB per privilege level.

In the same way as the Virtual TLB does, we can create different mappings for each privilege level. This would give a layout as we illustrated in Figure 7.1. There we expose the principle for a user-supervisor privilege levels case. Two embedded target address

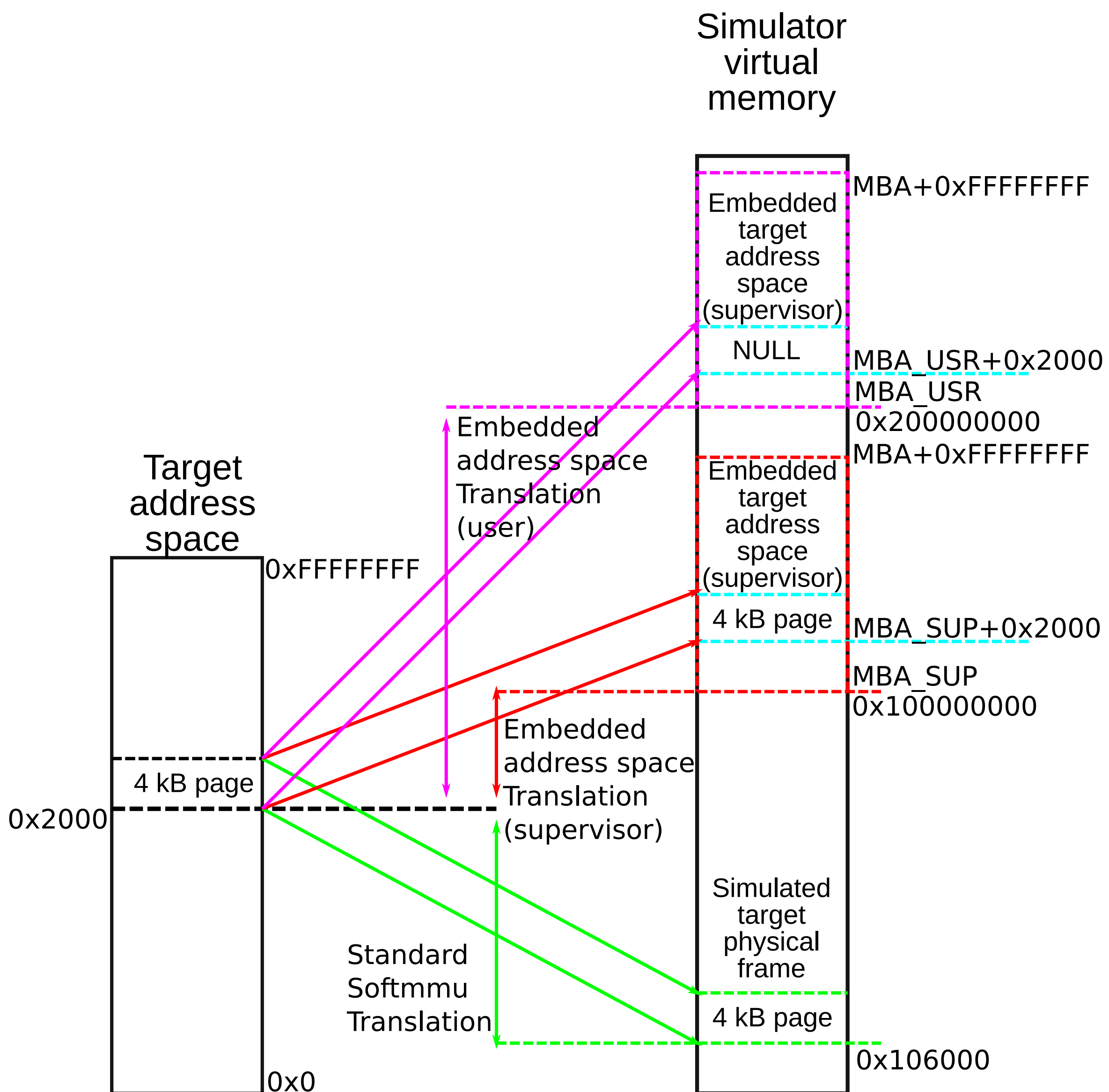


Figure 7.1: Simulator address space layout with multiple embedded target address space

space mappings exist, each one of them with their own offset. They also have their own corresponding user side handler on the simulator side. Now, if an access is made in target supervisor mode at target virtual address `0x2000`, only the host page table entry corresponding to `MBA_supervisor + 0x2000` will be filled and be aliased to a simulated target physical frame. If target user code tries to access the same address, the access will fail and trigger a fault as `MBA_user+0x2000` will still be empty, and go through the `softmmu_helper` to resolve the fault. This fault will then be percolated to the target, which will handle it. Thus solving this problem.

We already have an example of running multiple mappings in parallel, with in our example one for write accesses and one for read accesses. It shows no performance impact whatsoever, thus giving us good hope that creating one mapping per privilege level will work efficiently.

7.1.2 QEMU Multi threading support

Another item left for future work is the support of QEMU multi-threading *aka* MTTCG. Modern QEMU is indeed able to simulate each target CPU on a different thread, thus executing target code concurrently.

For now, our solution does not work in this mode, mainly due to the lack of synchronization points in our approach. Two main items need to be achieved to support MTTCG, first each thread need to have its own target address space mapping(s). Second, we need the module to be able to support multiple user threads, and to have its side effects targeted at the right threads. In particular if a target core flushes the TLB of another, we need to redirect the subsequent mapping flush to the right thread. Consequently, we also need to synchronize our threads so that a flush made from another thread does block the execution of said thread.

Overall, the multiple mapping approach has already been implemented. But for now, those threads can not communicate at kernel level, *i.e.* one thread may not flush another's mapping. And even if it did, execution of the thread would not be blocked, leading to incorrect behavior.

In the end, the remaining problems for MTTCG support are more technical than conceptual and are thus not high priority for now.

7.1.3 64-bit targets support

One important addition left to be done to our solutions is the support of 64-bit targets. Actually, this support can be envisioned in two different ways, one is applicable to both solutions, the other only to the HAV based one.

We already gave the basic idea of the solution for HAV in Chapter 4, but let us detail it further. The core idea is to give the target a full address space this time, by creating a second process parallel to that of the simulator. This principle is depicted in Figure 7.2.

There we can see the whole layout of the target's process. Obviously, the whole target address space is present, and can be seen in orange. But we also have to integrate two other items: the generated code cache, and a mechanism to call QEMU helpers.

For the code cache, we mostly need to find some free space in the process, and map it there. But for the helpers, it is more complicated. We can't just map the helpers as they often require other parts of the simulator and Linux syscalls. This means that in the end we would have to map, once again, almost all of the simulator, thus negating any advantage of our proposition. Alternatively, what we could do is to consider helpers themselves as syscalls. In this case, we could just map a simple syscall handler, which could then, in the same way as modern Linux does, switch page table to that of the simulator and execute the helpers there.

This method does expose a cost, since in many occasions, jumping from one target basic block to another implies a helper call. Considering that a basic block is rarely larger than a few (target) instructions, the overhead of causing one syscall per basic blocks could soon be fatal to the solution. What this essentially mean is that this solution would not be adapted to current QEMU. It would require a trace based code generator coupled with mulitple works to inline as many of the helper calls as possible (be it for memory accesses, or vectorized ISA). The idea being to reduce this communication costs as much as possible.

Also, some space will have to be found for the generated code cache and the syscall handler. Those are however relatively small, and few targets truly use all their address space. For example, now ARM OSes still usually do not use more than 2^{42} bytes of the AArch64 address space. And in the worst case, we can always imagine using a balloon driver to give us some memory, in the same way as paravirtualizers do, or even use some device area.

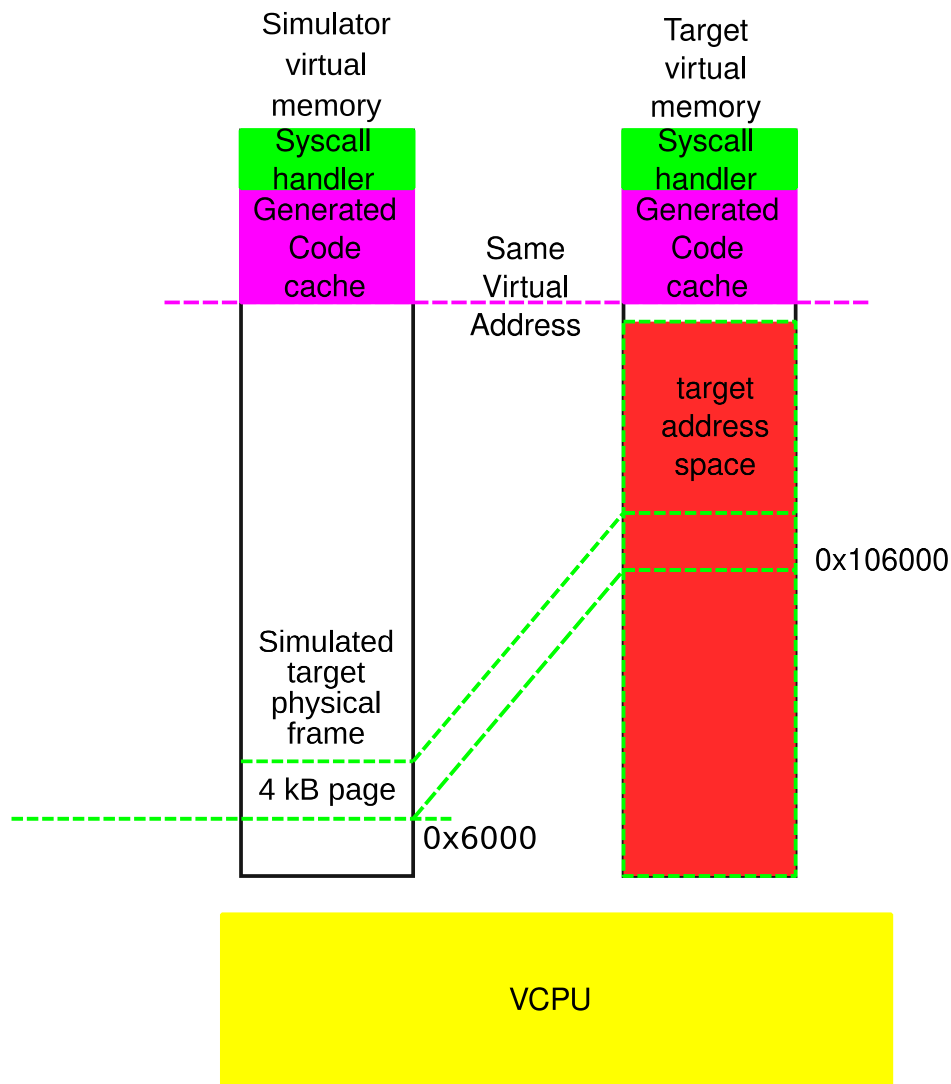


Figure 7.2: Simulator and target address space lying in two different processes

If using HAV is not possible, or if the second process solution were to be too slow (no trace based code generation) then another solution is possible. We can allocate sub parts of the target address space lazily. Indeed, the target will not constantly use its whole address space. And even then, most of what it uses will be relatively localized, locality being at the core of the notion of program.

To do so we modify our memory access inline code as can be seen in Figure 7.3. The idea here is that, instead of directly acquiring a constant Mapping Base Address, we have to lookup an array containing, for each target zone, the corresponding mapping on the host. For the rest, the simulator layout is then quite similar to Figure 7.1, except that instead of having user-supervisor mappings, we have one mapping per target memory area.

To make the lazy allocation, we can prefill the array with a special MBA. This MBA will be made so that any access to it will trigger a fault. In its own special handler it will then allocate a new MBA for the memory area of our faulty address, and then alias the faulty address using QEMU `softmmu_helper` as usual.

This method's main advantage is that it does not require any hardware support, or any

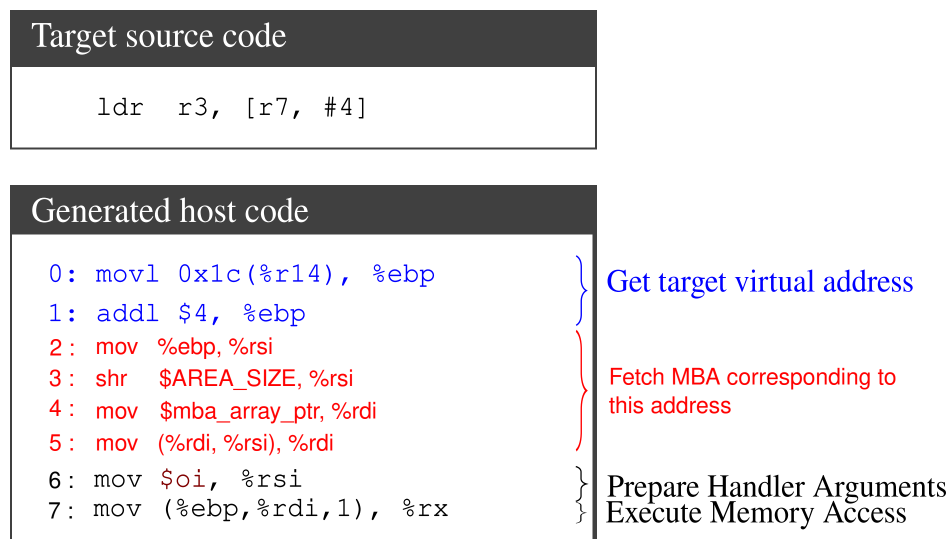


Figure 7.3: Fast path with lazy 64-bit target approach

important simulator modification. Its main disadvantage is that it adds an indirection for every memory access, thus reducing the interest of the method.

7.1.4 Handling write accesses with in kernel internal faults handling

Finally, our last challenge is to generalize in kernel fault handling support for the Linux kernel module design to write accesses. For now, this design is only able to manage read-accesses in kernel. Most of the problems currently come from the limited ability to communicate between the simulator and the Linux kernel module.

Most of the problem is that, in its current state, QEMU (when using TCG at least) is not thought with the use of a LKM in mind. Sharing data with the LKM in particular is fairly complicated. Structures used by QEMU are often complex, with many dependencies, which makes simply copying them to the module impractical. Even extracting them at runtime can get fairly complicated. The result is that, overall extracting the code to make a target page walk is not as straightforward as one could hope.

Managing writes also imply even more communications between the Linux kernel module and the simulator. In particular, whenever a write to target code is made, it must be forwarded to QEMU so that it may update its code cache. This in turn implies retrieving updates to the state of each target physical frame in QEMU. While this would seem easy at first, the code to do so in QEMU is completely Virtual TLB-centric. Essentially pages to be protected will often be passed as virtual addresses, and not necessarily when one would expect.

Another part of the problem comes from fitting the solution in Linux. The VMA handlers were thought to answer mostly three cases: find a free physical frame for a copy on write address, retrieve a swapped out page from disk, or retrieve a page from disk from a memory mapped file. More generally, an "empty" memory address space, which is made to point directly to physical frames from another mapping is not prepared for either. At the time we are in the handler, we only have a read access semaphore for the page table. In other words, if we try to modify the page table directly in handler, chances are that we may collide with other kernel process reading it. Releasing and taking back the semaphore is not a solution either. For read-accesses only, the solution can still work, but will crash randomly. To avoid

those problems, some modifications will probably have to be done in Linux too, although this will probably only be about adding an option to have write-semaphore protected page faults.

Overall, to continue forward we will mostly have to rework QEMU code so as to be able to easily reuse code in kernel. Structures will have to simplified for easier access, and to keep the fault handler as short and fast as possible.

On a broader view, we could also further adapt the module and simulator so as to simulate devices directly in kernel. A perfect example would be to simulate framebuffers directly in kernel, with a thin layer to adapt target framebuffer calls, to that of the host. Doing so would both avoid round trips between the simulator and kernel, but also avoid most of the simulation logic, since the kernel can really draw directly. This is left as an interesting perspective for future works.

Bibliography

[AEGS01] E.R. Altman, K. Ebcioğlu, M. Gschwind, and S. Sathaye. Advances and future challenges in binary translation and optimization. *Proceedings of the IEEE*, 89(11):1710–1722, November 2001.

[Amd67] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[BBM+12] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–348, 2012.

[BBTV15] Darrell Boggs, Gary Brown, Nathan Tuck, and KS Venkatraman. Denver: Nvidia's first 64-bit arm processor. *IEEE Micro*, 35(2):46–55, 2015.

[BDB11] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 46(4):41–52, 2011.

[BDE+03] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigel Zemach. Ia-32 execution layer: a two-phase dynamic translator designed to support ia-32 applications on itanium®-based systems. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 191. IEEE Computer Society, 2003.

[Bel05a] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[Bel05b] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[BEvKK+11] Igor Böhm, Tobias JK Edler von Koch, Stephen C Kyle, Björn Franke, and Nigel Topham. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *ACM SIGPLAN Notices*, volume 46, pages 74–85. ACM, 2011.

[BFT10] Igor Böhm, Björn Franke, and Nigel Topham. Cycle-accurate performance modelling in an ultra-fast just-in-time dynamic binary translation instruction set simulator. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 1–10. IEEE, 2010.

[BZA12] Derek Bruening, Qin Zhao, and Saman Amarasinghe. Transparent dynamic instrumentation. *ACM SIGPLAN Notices*, 47(7):133–144, 2012.

[CCL+14] Henri-Pierre Charles, Damien Couroussé, Victor Lomüller, Fernando A Endo, and Rémy Gauguey. degoal a tool to embed dynamic code generators into applications. In *International Conference on Compiler Construction*, pages 107–112. Springer, 2014.

[CK94] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 128–137, 1994.

[CVERL02] Cristina Cifuentes, Mike Van Emmerik, Norman Ramsey, and Brian Lewis. Experience in the design, implementation and use of a retargetable static binary translation framework. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2002.

[CWH+14] Chao-Jui Chang, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, and Pen-Chung Yew. Efficient memory virtualization for cross-isa system mode emulation. In *ACM SIGPLAN Notices*, volume 49, pages 117–128. ACM, 2014.

[DB00] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: Less is more. *ACM SIGOPS Operating Systems Review*, 34(5):202–211, 2000.

[DCHC11] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. Pqemu: A parallel system emulator based on qemu. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 276–283. IEEE, 2011.

[DGB+03] James C Dehnert, Brian K Grant, John P Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing tm software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, pages 15–24. IEEE Computer Society, 2003.

[DM10] Arnaldo Carvalho De Melo. The new linux "perf" tools. In *Slides from Linux Kongress*, volume 18, 2010. `http://vger.kernel.org/~acme/perf/lk2010-perf-acme.pdf`.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM.

[EA97] Kemal Ebcioğlu and Erik R. Altman. Daisy: Dynamic compilation for 100 *SIGARCH Comput. Archit. News*, 25(2):26–37, May 1997.

[ECC15] Fernando A Endo, Damien Couroussé, and Henri-Pierre Charles. Towards a dynamic code generator for run-time self-tuning kernels in embedded applications. In *4th International Workshop on Dynamic Compilation Everywhere*, 2015.

[Gar03] Timothy Garnett. *Dynamic optimization if IA-32 applications under DynamoRIO*. PhD thesis, Massachusetts Institute of Technology, 2003.

[GF06] Andreas Gal and Michael Franz. Incremental dynamic code generation with trace trees. Technical report, Citeseer, 2006.

[GFP09] Marius Gligor, Nicolas Fournel, and Frédéric Pétrot. Using binary translation in event driven simulation for fast and flexible MPSoC simulation. In *Proceedings of the 7th IEEE/ACM international conference on hardware/software codesign and system synthesis*, pages 71–80. ACM, 2009.

[GLS+17] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems*, pages 161–176. Springer, 2017.

[HHC+16] Ding-Yong Hong, Chun-Chen Hsu, Cheng-Yi Chou, Wei-Chung Hsu, Pangfeng Liu, and Jan-Jan Wu. Optimizing control transfer and memory virtualization in full system emulators. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):47, 2016.

[HHY+12] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. Hqemu: a multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 104–113. ACM, 2012.

[HPF13] M. Hamayun, F. Pétrot, and N. Fournel. Native simulation of complex vliw instruction sets using static binary translation and hardware-assisted virtualization. In *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, pages 576–581, Jan 2013.

[JRHC95] Lizy Kurian John, Vinod Reddy, Paul T. Hulina, and Lee D. Coraor. Program balance and its impact on high performance risc architectures. In *First IEEE Symposium on High-Performance Computer Architecture*, pages 370–379. IEEE, 1995.

[KCB07] Minhaj Ahmad Khan, Henri-Pierre Charles, and Denis Barthou. Reducing code size explosion through low-overhead specialization. In *The 11th Workshop on Interaction between Compilers and Computer Architectures*, page 82. Citeseer, 2007.

[LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018.

[May87] C. May. Mimic: A fast system/370 simulator. *SIGPLAN Not.*, 22(7):1–13, July 1987.

[MCE+02] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[MFP11] L. Michel, N. Fournel, and F. Pétrot. Speeding-up simd instructions dynamic binary translation in embedded processor simulation. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–4, March 2011.

[Mic14] Luc Michel. *Contributions à la traduction binaire dynamique: support du parallélisme d'instructions et génération de traducteurs optimisés*. PhD thesis, Université de Grenoble, 2014.

[MS08] Darek Mihocka and Stanislav Shwartsman. Virtualization without direct execution or jitting: Designing a portable virtual machine infrastructure. In *1st Workshop on Architectural and Microarchitectural Support for Binary Translation in ISCA-35, Beijing*, page 32, 2008.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[PGH+11] F. Pétrot, M. Gligor, M.-M. Hamayun, Hao Shen, N. Fournel, and P. Gerin. On mpsoc software execution at the transaction level. *Design Test of Computers, IEEE*, 28(3):32–43, May 2011.

[PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspot tm server compiler. In *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1*, pages 1–1. USENIX Association, 2001.

[PY16] Louis-Noël Pouchet and T Yuki. Polybench/c 4.2, 2016. `https://sourceforge.net/projects/polybench/`.

[Rau78] B. Ramakrishna Rau. Levels of representation of programs and the architecture of universal host machines. *SIGMICRO Newsl.*, 9(4):67–79, November 1978.

[RBMD03] M. Reshadi, N. Bansal, P. Mishra, and N. Dutt. An efficient retargetable framework for instruction-set simulation. In *Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on*, pages 13–18, Oct 2003.

[RRD17] Simon Rokicki, Erven Rohou, and Steven Derrien. Hardware-accelerated dynamic binary translation. In *Proceedings of the Conference on Design, Automation & Test in Europe*, pages 1062–1067. European Design and Automation Association, 2017.

[RS97] Alasdair Rawsthorne and Jason Souloglou. Dynamite: A framework for dynamic retargetable binary translation. Technical report, Tech. Report UMCS 97-3-2, University of Manchester, Manchester, UK, 1997.

[RSR16] Alvise Rigo, Alexander Spyridakis, and Daniel Raho. Atomic instruction translation towards a multi-threaded qemu. In *ECMS*, pages 587–595, 2016.

[SBP16] Guillaume Sarrazin, Nicolas Brunie, and Frédéric Pétrot. Virtual prototyping of floating point units. In *Proceedings of the 2016 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, pages 1:1–1:6, New York, NY, USA, 2016. ACM.

[SFGP15] Guillaume Sarrazin, Nicolas Fournel, Patrice Gerin, and Frédéric Pétrot. Simulation native basée sur le support matériel à la virtualisation : cas des systèmes many-cœurs spécifiques. *Technique et Science Informatiques*, 34(1-2):153–173, 2015.

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

[SWF16] Tom Spink, Harry Wagstaff, and Björn Franke. Hardware-accelerated cross-architecture full-system virtualization. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(4):36, 2016.

[TGN95] Marc Tremblay, Dale Greenley, and Kevin Normoyle. The design of the microarchitecture of ultrasparc-i. *Proceedings of the IEEE*, 83(12):1653–1663, 1995.

[WLW+15] Zhe Wang, Jianjun Li, Chenggang Wu, Dongyan Yang, Zhenjiang Wang, Wei-Chung Hsu, Bin Li, and Yong Guan. Hspt: Practical implementation and efficient management of embedded shadow page tables for cross-isa system virtual machines. In *ACM SIGPLAN Notices*, volume 50, pages 53–64. ACM, 2015.

[WR96] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. *SIGMETRICS Perform. Eval. Rev.*, 24(1):68–79, May 1996.

[XTMA15] Tong Xin, Koju Toshihiko, Kawahito Motohiro, and Moshovos Andreas. Optimizing memory translation emulation in full system emulators. In *ACM Transactions on Architecture and Code Optimization (TACO)*, 2015.

[ZT00] Cindy Zheng and Carol Thompson. Pa-risc to ia-64: Transparent execution, no recompilation. *Computer*, 33(3):47–52, March 2000.

Résumé

Dans cette thèse nous nous intéressons à l'accélération des accès mémoire dans la traduction binaire dynamique. Pour cela, nous nous basons sur des méthodes dont la principale finalité est de gérer l'espace mémoire de la cible avec le matériel de l'hôte. Deux grandes méthodes pour cela ont été exploré, l'une basé sur la support matériel à la virtualisation, et l'autre sur un module Linux.

Dans le cas du support matériel à la virtualisation, nous avons utilisé le simulateur comme un invité spécifique. Celui ci jouant un rôle analogue à celui d'un OS, en plus de son rôle de simulateur, pour la cible. En particulier il se charge de lui créer un espace d'adressage enchevêtré, qui puisse être utiliser directement, sans simulation logicielle de la gestion de la mémoire virtuelle.

Dans le cas de la méthode basée sur un module Linux, les mêmes finalités sont poursuivi. Mais le simulateur continue de fonctionner comme un processus normal. En revanche, il possède désormais un module compagnon, avec lequel il peut communiquer au travers d'ioctl. Ce module est chargé de manipuler la gestion de la mémoire virtuelle de l'hôte et ce afin de créer un espace d'adressage enchevêtré pour la cible.

Ces méthodes ont été implémenté dans Qemu et Linux et mène à des gains de performances significatifs.

Abstract

In this thesis we are interested in the acceleration of memory accesses in dynamic binary translation. For this, we base ourselves on methods whose main purpose is to manage the target's address space with the host's hardware. Two main methods for this have been explored, one based on hardware assisted virtualization, and the other on a Linux module.

In the case of hardware assisted virtualization, we used the simulator as a specific guest. This one playing a role similar to that of an OS, in addition to its role of simulator, for the target. In particular, it is responsible for creating an enmbedded address space that can be used directly, without software simulation of an MMU.

In the case of a method based on a Linux module, the same purpose is pursued. But the simulator continues to operate as a normal process. On the other hand, it now has a companion module, with which it can communicate through ioctl. This module is responsible for manipulating the host's virtual memory management to create an embedded address space for the target.

These methods have been implemented in Qemu and Linux and lead to significant performance gains.