Intel Processor Trace for Reconstructing Guest Control Flow in a Dynamic Binary Translator

Robbie John Wallace
180013940

University of
St Andrews

Project Supervisor: Tom Spink

School of Computer Science
University of St Andrews
6 January 2023

Abstract

Intel Processor Trace [15] (Intel PT) is a recent addition to Intel CPUs. Intel PT is a hardware-based system for tracing the execution flow of programs. Dynamic Binary Translation is a form of binary recompilation. That is taking a binary program written in a guest Instruction Set Architecture (ISA), and translating it to a target ISA. This work explores the possibility of combining Intel PT with QEMU [23] (an open-source DBT) to use it to generate a program trace for a guest program written in a non-Intel ISA. After demonstrating this is possible, the solution is evaluated against traditional trace generation methods through the use of the SPEC CPU 2017 [29] benchmark suite. Though the evaluation shows that traditional software is faster, a redesign is proposed, which should make the Intel PT tracing method the faster approach.

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is 19,468 words long, including project specification and plan.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Acknowledgements

First, I would like to thank my supervisor Tom Spink for his guidance. This project has fun, but at times very challenging and without Toms support it would have been significantly harder. I also want to thank the numerous lecturers and tutors I have had throughout my five years studying at St Andrews, without their teachings I would not be programmer I am now.

Completing this project or even my degree would not have been possible without the constant support that caffeine provides. I must thank Evan Church for inventing the drip coffee machine. During the trying semester in which I completed this project, the espresso machine in the department has been temperamental at best. However, the department's old drip coffee machine has stood firm and provided Coffee when other devices have failed.

Finally, I thank my friends and family, without their constant support, I would not have been able to complete not only this project but also my degree. Most importantly, I would like to thank my mom, who has also been there to correct my, at times, dubious understanding of the English language.

Contents

1 Introduction

Intel has recently included a technology called "Processor Trace" [15] (Intel PT) which is a hardware-based system for tracing the execution flow of programs. This technology works to quickly generate a trace for an application running on the machine, which can be used for debugging and profiling information.

A Dynamic Binary Translator [27] (DBT) takes a program written in one Instruction Set Architecture (ISA), and allows it to execute on another, by translating blocks of instructions on demand from the guest ISA to the host ISA.

This project will investigate if a host ISA execution trace generated using Intel PT can be translated into the equivalent execution trace of a guest ISA. Instead of writing an entire DBT from scratch to perform this investigation QEMU, an open-source machine emulator and virtualizer will be used.

1.1 Objectives

Primary

- Implement basic block trace generation in QEMU without the use of Intel PT
- Implement basic block trace generation in QEMU using Intel PT
- Analyse and Compare the two implementations

Secondary

- Implement the ability to dynamically slow down the execution of code to prevent Intel PT from generating too much data

1.2 Software Engineering Process

The development of this project took an agile approach. Tracing methods were developed iteratively with each iteration building on the previous. After the development of each version, a small evaluation was performed to understand what worked well and what didn't. This approach worked well and was mostly required as the requirements of this project were to explore the possibility of using Intel PT. Only after insights could be gained from naive approaches could more sophisticated methods be developed.

A benefit of developing numerous tracing methods allows various methods to be used to test against themselves. The traces were also followed by hand for sections of smaller programs to ensure they are correct. However, even for a simple c program with an empty main, the resulting trace is over 20,000 blocks long, due to dynamic linkage, and other C runtime aspects.

1.3 Ethics

This project has raised no ethical concerns. The self-assessment form is included in the appendix A.

2 Context Survey

2.1 Instruction Set Architecture

At a high level, the Instruction Set Architecture (ISA) is the boundary between software and hardware[13]; it defines the abstract model of a computer and how a programmer can interact with the CPU. An ISA refers to a set of instructions which can be used to interact directly with the hardware. Programmers do not usually write programs written with these instructions. Instead, they write programs written in an assembly language [14], or asm for short.

These assembly instructions do not run directly on the hardware. It is machine code which runs on the hardware. Each instruction in an assembly language typically has a one-to-one mapping to a given machine instruction. Meaning that each ISA has an assembly language linked to it. The assembly code is converted into executable machine code by an assembler. This executable machine code is called object code.

Unlike higher-level programming languages such as C, assembly language is entirely machine-dependent. A program written in ARM assembly will only run on an Arm CPU, and likewise, a program written in Intel x86 assembly will only run on an Intel CPU.

The hardware dependency can be more specific than Intel assembly can only run on Intel CPUs. For example, modern Intel CPUs have vector registers, a program which utilises these will not run on an older Intel CPU as those vector registers did not exist. This hardware specificity is in direct contrast with higher-level languages such as C, as a program written in C can be compiled into many different ISAs for execution on specific machines.

2.2 Basic Blocks & Program Traces

When dealing with assembly, it is often convenient to think about programs not as a series of instructions but as a series of basic blocks. A basic block is a sequence of instructions with a single entry point, containing no branches, with a single exit branch [10]. That is a basic block is a linear sequence of instructions that ends with a branch to another basic block.

```
movq $12, %eax
movq $30, %ebx
addq %ebx, %eax
jmp 0x1248
```

Figure 1: Basic Block Example

Figure 1, above, shows a single basic block. At the start of this basic block, the values 12 and 30 are moved into registers a and b. Then the result of a + b is moved into register a. This block ends with a jump to some other location. Ignoring the specific instructions being executed, the result of executing the basic block is to move 30 into register b and 42 (12 + 30) into register a.

As an optimisation, a compiler could re-write this basic block to set registers a and b to 42 and 30. The block would no longer need the `addq` instruction. This is one of the key benefits of thinking about programs as a sequence of basic blocks, not instructions. If the effect a basic block has on the state of a CPU is preserved (in this case, setting registers a and b to 42 and 30), it doesn't matter how the basic block achieves this effect; the result of executing the program will remain the same.

```c
int calculate_sum(
    int limit
) {
    int sum = 0;

    for(int i = 0; i < limit; i++) {
        sum += i;
    }

    return sum;
}
```

(a) Input C Code

```
1  calculate_sum(int):
2  pushq   %rbp
3  movq    %rsp, %rbp
4  movl    %edi, -20(%rbp)
5  movl    $0, -4(%rbp)
6  movl    $0, -8(%rbp)
7  jmp     .L2
8  .L3:
9  movl    -8(%rbp), %eax
10 addl    %eax, -4(%rbp)
11 addl    $1, -8(%rbp)
12 .L2:
13 movl    -8(%rbp), %eax
14 cmpl    -20(%rbp), %eax
15 jl      .L3
16 movl    -4(%rbp), %eax
17 popq    %rbp
18 ret
```

(b) Output Assembly Code

Figure 2: Translation of a Simple Function

Figure 2 above shows a C function on the left with its translation in assembly on the right. In the output assembly, there are four basic blocks: Block 0 (B0) is lines 1 to 7 (this block sets sum and i to zero); Block 1 (B1) is lines 8 to 11 (this block adds i to sum and then increments i by one); Block 2 (B2) is lines 12 to 15 (this block checks if i is still less than the limit); lastly, there is Block 3 (B3) lines 16 to 18 (This block returns the value of sum).

One thing that may appear strange about these blocks is that B1 does not end with a jump instruction. Recall that a basic block must only contain one entry point. As such, lines 8 to 15 cannot be a single block. The jump instruction at line 7 branches to line 12. So, though B1 does not end with a jump, it can be thought of as containing an unwritten branch to the next block, B2.

Analysing the blocks in Figure 2, it is clear that they do not form a neat path; instead, they form a directed cyclic graph. B0 always branches to B2. B2 branches to either B1 if i is still less than the limit, or it branches to B3 if i is not less than the limit. B3 branches to any number of other blocks as it makes use of the `ret` instruction. This instruction returns control flow back to the location this function was called from [19].

As a program executes, it can be thought of as walking this graph of basic blocks. The path that is generated from this walk can be called a program trace. From this trace, it is possible to work out exactly what instructions were executed and in what order from executing this program. If an input limit of 2 were given to the code in Figure 2, the following trace would be generated: B0, B2, B1, B2, B1, B2, B3.

2.3 Dynamic Binary Translation

Binary Translation [27] is the process of translating object code written in a guest Instruction Set Architecture (ISA) to a target or host ISA. For example, if there is object code written for an Arm (guest) CPU and a user wants to execute that on an Intel x86 (target/host) CPU, it needs to be translated.

Binary translation is not an easy process. It would be simpler if a user could re-compile the original program into the target ISA. However, the original program may not exist, or the program could have been written in assembly code to begin with. In this situation, the only possibility is to use Binary Translation.

There are two main methods for implementing Binary Translation. Firstly, there is Static Binary Translation (SBT). In SBT, the entire program is translated into the target ISA, which can then be executed. While this may appear simple, there are a lot of difficulties with SBT. For instance, there may not be a clean one-to-one mapping from each guest instruction to a target instruction.

Early ARM CPUs did not have divide instructions, so to translate an Intel x86 `div` instruction, an entire function needs to be generated. Then there are situations where different ISAs have differing amounts of registers. How could a write to a register that doesn't exist on the target machine be translated?

In theory, these problems could be overcome by having a very complex translation process. However, it leaves out one of the biggest problems, which is computed jumps. A computed jump is a jump to a location which is computed at runtime. For example, a program could contain the following copmuted jump `jmp eax`.

Since the translated code will not be a one-to-one mapping (one input instruction becomes one output instruction), the location of this jump will be invalid in the translated code. The only possibility to resolve this problem is to store a table mapping from guest instruction location to target instruction location. Then when a computed jump is executed, a lookup can be made in this table to determine the translated location of the jump.

SBT is becoming more and more complex and is essentially building a runtime system inside the translated output. So, instead of building a runtime inside every translated program, it would be easier to use a runtime system to execute the original guest program. This idea leads to the second method of Binary Translation, Dynamic Binary Translation (DBT).

In DBT, code is translated only when it is needed through the use of a runtime system. This runtime generates target code, typically one basic block at a time. When a jump is executed, the runtime system looks up the location of this jump. If it has been translated into target code, the DBT can begin executing it. Otherwise, it will translate the required code and then begin execution.

2.4 QEMU

Qemu [23] is an open-source machine emulator and virtualizer which makes use of DBT to execute guest object code on a target machine. QEMU has the ability to run user mode emulation or kernel mode emulation. User mode emulation is the process of emulating user programs (programs written to run on an operating system, e.g. Linux). Where kernel mode emulation is the process of emulating an entire operating system, this project will not focus on kernel emulation.

QEMU is an extremely vast program with an abundance of features, and to explain them all would take far too much time. Instead, a high-level overview of QEMU will be given, and if more spesific knowledge is required in later stages of the report, it will be provided where needed. To understand the choices behind QEMUs design, it is important to understand how much it does. QEMU supports over 20 different ISAs for guest-level emulation [24].

It would be exceedingly difficult to write a program which can translate from any one of these ISAs to any other one (over 190 different combinations). To achieve this goal, QEMU makes use of an intermediate representation (IR). Meaning it translates all guest ISAs into a single IR and then translates this IR into the target ISA; thus, to implement a new ISA into QEMU, all that is required is to implement methods for translating it to and from this IR.

The core region of QEMU, which performs the translation and execution of guest ISA is referred to as the Tiny Code Generator (TCG). At its core the TCG contains the main execution loop of QEMU. This loop takes in an address of a basic block in the guest program. It performs a lookup to check if this basic block has been transled. If it has not been translated then it translates the basic block into a translated block (TB). The translated block is then executed. After execution, the translated block returns the address of the next basic block it wants to execute. The loop can then restart with this new address. This process is started with the first basic block in the guest program and continues until the guest program exits.

To ensure that the translated program produces the same result as the original guest program, QEMU needs to ensure that the effect of executing a translated block on the CPU is the same as its original guest basic block would have on the CPU (see basic block section for an explanation as to why that is needed). However, the target CPU is a different CPU from the guest CPU.

To solve the problem of difference between CPUs QEMU introduces the CPUState struct. For each ISA supported by QEMU, it has a corresponding CPUState struct that contains an entry for all registers and any other CPU spesific state that needs to be maintained (e.g. error flags, etc...).

This CPUState struct can be passed into each translated block. The TB then needs to ensure that before exiting, it updates this struct to represent the effect the original guest basic block would have had on a real guest CPU. This helps solve a lot of the problems associated with Binary Translation. For example, if there is any mismatch between the number and types of registers supported by the guest and target ISAs that doesn't matter, as it is only what is contained within the CPUState struct which matters.

A final piece of terminology that is needed to understand QEMU is guest vs target addresses. The program which QEMU is exeucting is stored within target/host memory. However, when that program performs a memory operation, it is expecting to be reading and writing to guest memory. QEMU has to perform a translation to get the target address for a given guest address.

When dealing with QEMU the term guest address refers to an address in the memory of the guest CPU, which does not really exist. Target or host address refers to the real address in memory that is being accessed for a given guest address.

2.5 Intel Processor Trace

Intel Processor Trace (Intel PT) [15] is an extension to the Intel architecture that allows for the collection of information about a program's execution with minimal performance impact. The information which can be collected ranges from control flow to power and timing information. Information is generated in highly compressed binary packets.

The collection of Intel PT data was introduced into the Linux Perf API in version 4.2 of the kernel [20]. Through this API a user has the ability to start and stop the collection of Intel PT data through system calls. The data is saved into buffers which can be mapped into user memory [21].

2.6 Related Work

Due to Intel Processor Trace being a relatively new introduction to the Intel architecture, there isn't a significant amount of related work. Most work in this area regards improving QEMUs tracing abilities using software alone, and more specifically, most research looks into improving QEMUs kernel level tracing abilities. In contrast, this work looks at guest level tracing only.

For instance, some work in this area is 'Dynamically Instrumenting the QEMU Emulator for Linux Process Trace Generation with the GDB Debugger' [17]. This paper looks at bringing the Gunu Debugger into QEMU to increase the performance of trace generation whilst also providing features such as register value logging and kernel level tracing.

There does not appear to be any work which looks at using Intel PT in a DBT. However, papers like the one described above provide useful functionality which could be brought into this work in future, such as kernel level tracing.

3 Design & Implementation

This chapter describes the design and implementation of the various tracing implementations created and how the insights gained from initial implementations influenced later versions.

Tracing methods which utilise Intel Processor Trace (Intel PT) in some way are referred to as hardware tracing methods, as they require the Intel PT hardware features to work. Versions which do not use Intel PT, and instead use traditional software methods to generate a trace are referred to as software tracing methods.

3.1 Software Version I

The first software tracer is implemented using the most simple approach possible. That is to log the original guest address of each translated block (TB) prior to executing it.

Recall from the context survey section that QEMU works by translating each guest basic block into a translated block. These blocks are executed in QEMUs execution loop. To generate a trace of the guest program, QEMU could be modified to log the guest address of each translated block prior to executing it.

Thankfully the translation block struct has this guest address as an attribute already and does not need to be modified. So, all that is needed to implement this tracer is to add the guest address logging. To implement this, the `cpu_tb_exec()` function is modified. This function lies at the core of QEMUs execution loop; it takes in a translated block and executes it.

To generate a trace, this function was modified by adding a call to `log_guest_block_exec()`. The `log_guest_block_exec()` function is provided with the guest address of the translated block, which it will save to a 'trace.txt' file. The result of this modification can be seen in Figure 3.

```
// cpu-exec.c
static inline TranslationBlock * QEMU_DISABLE_CFI
cpu_tb_exec(CPUState *cpu, TranslationBlock *itb, int *tb_exit)
{
    CPUArchState *env = cpu->env_ptr;
    uintptr_t ret;
    TranslationBlock *last_tb;
    const void *tb_ptr = itb->tc.ptr;

    // ...

    log_guest_block_exec(itb->pc);
    ret = tcg_qemu_tb_exec(env, tb_ptr);

    // ...

    return last_tb;
}
```

Figure 3: cpu_tb_exec source code

There is one problem that needs to be solved before this implementation is complete, and that is direct block chaining. Direct block chaining allows translated blocks to jump directly to other translated blocks during execution without exiting out of translated code and returning to the QEMU execution loop.

QEMU implements direct chaining because it can lead to massive performance benefits in most programs. For instance, imagine a program containing a loop. The basic blocks which make up that loop are going to be executed multiple times. Without direct chaining after executing each translated block, execution must return to QEMU to search for the next translated block, incurring an overhead. Direct chaining removes this overhead.

What this means in practice is that though `tcg_qemu_tb_exec()` (seen above in Figure 3) is only called with a single translated block, it can lead to the execution of any number of translated blocks. This will cause large gaps to exist in the output trace if nothing is done to correct this problem.

To keep this implementation as simple as possible, the chosen solution is to disable direct block chaining. QEMU makes use of two different methods for achieving direct block chaining: 'lookup_and_goto_ptr', which will be referred to as computed chaining and 'goto_tb + exit_tb', which will be referred to as patch chaining [25].

In computed chaining, when a translated block has finished executing, a call is made to the `lookup_tb_ptr()` helper function. This function shown in Figure 4 will check if the next guest basic block has been translated. If the block has been translated, it returns a pointer to the relavent translation block. If it does not exist, it will return a pointer to the epilogue code. This epilogue code returns execution to the `cpu_tb_exec()` function discussed earlier.

Computed chaining may be seen when a basic block contains a computed jump. The translated block will perform the calculation to get the location of this jump. It will then call this helper method. If the location of the computed jump has already been translated, the helper returns a pointer to this translated code, and execution of that code can begin. Otherwise, the translated block must return to QEMU.

There are two options to deal with computed chaining. One option is to make the helper function only return the epilogue pointer, forcing execution back to `cpu_tb_exec()`. The second solution is to modify the helper to log the guest address of translated blocks it finds.

The second approach is preferable as the first approach would render the `lookup_tb_ptr()` helper function pointless. Whilst the second approach doesn't do this and could still give the performance of using it. Thus the second option is the one that is implemented. This implementation is achieved by adding the call to `log_guest_block_exec()` to the helper function, shown below in Figure 4.

```
// cpu-exec.c
const void *HELPER(lookup_tb_ptr)(CPUArchState *env)
{
    CPUState *cpu = env_cpu(env);
    TranslationBlock *tb;
    target_ulong cs_base, pc;
    uint32_t flags, cflags;

    // ...

    tb = tb_lookup(cpu, pc, cs_base, flags, cflags);
    if (tb == NULL) {
        return tcg_code_gen_epilogue;
    }

    log_guest_block_exec(tb->pc);
    return tb->tc.ptr;
}
```

Figure 4: lookup_tb_ptr code

With computed chaining solved, path chaining must be dealt with. This method is more complex. It works by modifying the assembly of a translated block after the initial translation. It occurs when a basic block ends with a direct jump to another basic block

In theory, the translation of the first basic block can contain a direct jump to the translation of the second basic block. If this could be achieved, it would offer significant performance improvements as no method calls are needed to go from one block to another. However, the translation of the second basic block may not exist when the first basic block has been translated.

To allow this direct jump, QEMU modifies the translation of the first basic block after the second basic block has been translated. Meaning that when the first translated block has finished executing, it will initially return to QEMU to cause the second translated block to be generated. After this has been completed, the first translated block can be patched to contain a direct jump to the second translated block. Thus, if the first translated block is ever executed again, it does not need to return to QEMU.

The low-level details of how this is implemented are quite complex, and it would be beneficial if it did not need to be touched. Thankfully, QEMU allows this method to be disabled by setting flags on the CPU Struct. To disable patch chaining, the relavent flag is set on the CPU. This can be seen in Figure 5 below.

```
// linux-user/main.c
int main(int argc, char **argv, char **envp)
{
    struct target_pt_regs regs1, *regs = &regs1;
    struct image_info info1, *info = &info1;
    struct linux_binprm bprm;
    TaskState *ts;
    CPUArchState *env;
    CPUState *cpu;

    // ...

    cpu = cpu_create(cpu_type);
    cpu->tcg_cflags |= CF_NO_GOTO_TB;

    // ...
}
```

Figure 5: main code

By disabling the patch chaining method, the first software tracing method has been successfully implemented. A small evaluation of it is performed. Doing so will provide insight into what worked well and what didn't. These insights can be used to influence the design of the following tracing methods.

To evaluate this method, a series of SPEC CPU [29] benchmarks were run. These benchmarks are explored in more detail in the evaluation section. The results of running these benchmarks can be seen below in Figure 6.

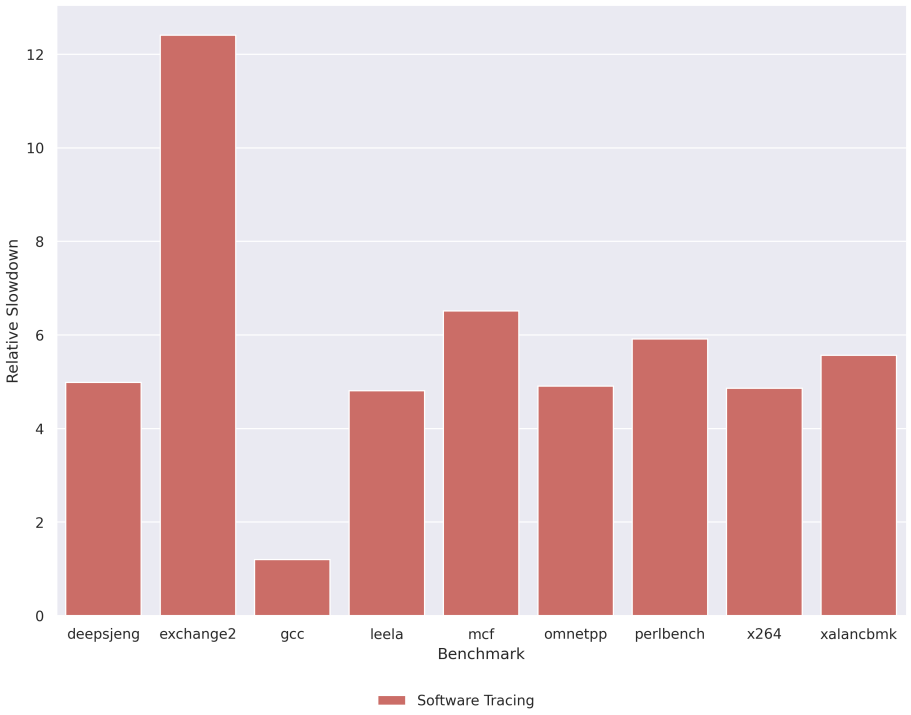


Figure 6: Relative slowdown on benchmarks from software tracing

Figure 6 above shows the effects software tracing had on the runtime of several benchmarks. The graph plots the relative slowdown for each benchmark. For example, looking at the benchmark deepsjeng it was five times slower to finish executing with tracing enabled than without.

Analysing the results in Figure 6, it shows that most benchmarks run somewhere between four to six times slower when generating a trace with this method. A slowdown of this magnitude is expected as direct chaining leads to massive performance benefits, and this tracing method disables the patch chaining method.

There are two outliers in these results, the gcc and exchange2 benchmarks. To start with for gcc when tracing is enabled there isn't a significant slowdown introduced. There are a few reasons for this. Firstly, the gcc benchmark is a very short running benchmark. Taking less than half a second to complete when running normally.

Due to the shortness of the program, a higher proportion of the runtime is spent starting up QEMU and then cleaning it up after the program has finished. Since the slowdown introduced by tracing only affects the portion of the runtime which deals with executing the guest program, the overall runtime is slowed down by a smaller proportion when compared to other benchmarks.

Secondly, gcc is a compiler and for the benchmark, it is given a simple C program to compile. What this means in practice is that the guest program is not running the same blocks of code many times. To explain, the simple C program only contains a single function call. This means the code in the compiler that deals with compiling function calls is only executed once.

As the gcc benchmark does not execute the same blocks many times, it will not benefit from the patch chaining method because that method only produces benefits when running the same block more than once. As such, this benchmark is not slowed down significantly by disabling patch chaining.

The other outlying result is the exchange2 benchmark. This benchmark is on the opposite end of the spectrum when compared with gcc. The results show that its runtime has slowed down significantly, even when compared to the other benchmarks. Reading the description of the exchange2 benchmark it states that this benchmark is used for generating non-trivial sudoku puzzles and importantly "it relies heavily on recursion (up to eight levels deep)" [9].

Since there is a large amount of recursion in the exchange2 benchmark, it means that the same basic blocks are going to get executed a lot of times. Coupled with the fact that this benchmark is long-running (over 30 seconds when running normally). This benchmark will benefit significantly from the patch chaining which has been disabled by this tracing method.

From these results, it can be concluded that keeping both forms of direct chaining enabled is paramount to ensuring there is not a significant performance decline in the runtime of QEMU.

3.2 Software Version II

The first software version has a significant slowdown when compared to QEMU running with no trace being generated. This slowdown comes from the lack of direct chaining.

Before implementing a hardware approach, an attempt at a better software approach is warranted for two reasons. Firstly, when comparing hardware to software, it would be better if the software approach was as good as it could be. Secondly, having a second software approach means the traces produced by both can be compared to ensure there are no bugs.

Based on the evaluation of the previous method, an area for improvement would be to re-enable direct chaining. This requires a re-work of the original approach, since logging translated block address inside the `cpu_tb_exec()` function will not work with chaining enabled.

What is needed is for the translated blocks themselves to signal when they start executing. Doing so would mean it doesn't matter if a block has been executed via direct chaining or from the `cpu_tb_exec()` function. This is possible through the use of QEMU helper functions.

QEMU allows for the creation of helper functions. These functions are regular C functions that can be called by translated code. An example of a helper function already seen is `lookup_tb_ptr()`, used by the computed chaining method. Helper functions are used for a wide array of features in QEMU. A common example is to implement a hardware feature that is not present in some architectures, vector registers, for example.

To implement this version, the `log_guest_block_exec()` helper function was created. This helper function (shown below in Figure 7) takes in the guest address of a translated block and logs it. One thing to note is that the HELPER macro appends 'helper' to the name of the given function. So while in Figure 7 the function name appears as `log_guest_block_exec()` in reality, it is `helper_log_guest_block_exec()`.

```
// arm/helper.c
void HELPER(log_guest_block_exec)(long unsigned pc) {
    log_guest_block_exec(pc);
}
```

Figure 7: log_guest_block helper code

With this helper function created, all that is left is to have it be called at the start of every translated block. This can be achieved by modifying QEMUs translation code. `gen_intermediate_code()` is a function which is called to translate a guest basic block into target code which is stored within the given translation block.

To add the desired functionality, a call to `helper_log_guest_block_exec()` needs to be inserted at the start of generated code. QEMU auto-generates a series of functions to add instructions to a translation block. These functions insert the given instruction at the end of the code stored within a given translation block.

Using these functions, a call to the desired helper function can be inserted. If this is done prior to all other code generation, the helper call will be the first instruction executed in every translated block. The result of this is shown below in Figure 8.

One final thing to note, since this helper is called within the translated code itself, it doesn't have access to the translation block struct. This means it cannot get the guest address attribute from that block. Instead, the value of the program counter (pc) register is supplied to the helper function. Since this is the first instruction executed in the translated block, the program counter value is equal to the guest address of this block.

```
// arm/translate.c
static inline void
add_log_guest_block_exec_to_tb(TranslationBlock *tb, target_ulong pc)
{
    TCGv_i64 tmp = tcg_temp_new_i64();
    tcg_gen_movi_i64(tmp, pc);
    gen_helper_log_guest_block_exec(tmp);
    tcg_temp_free_i64(tmp);
}

/* generate intermediate code for basic block 'tb'.  */
void gen_intermediate_code(
        CPUState *cpu, TranslationBlock *tb, int max_insns,
        target_ulong pc, void *host_pc
) {
    add_log_guest_block_exec_to_tb(tb, pc);

    // ...
}
```

Figure 8: log_guest_block translate code

This design now causes each translation block to call the `helper_log_guest_block_exec()` function when it begins executing. The `helper_log_guest_block_exec()` function logs the original guest address of that translation block. This will generate a complete trace of the program and also allows direct chaining to be re-enabled.

To test if this design is better than the previous software approach, it is evaluated against it. The results of running the same benchmarks as before with this method are shown in Figure 9 below.

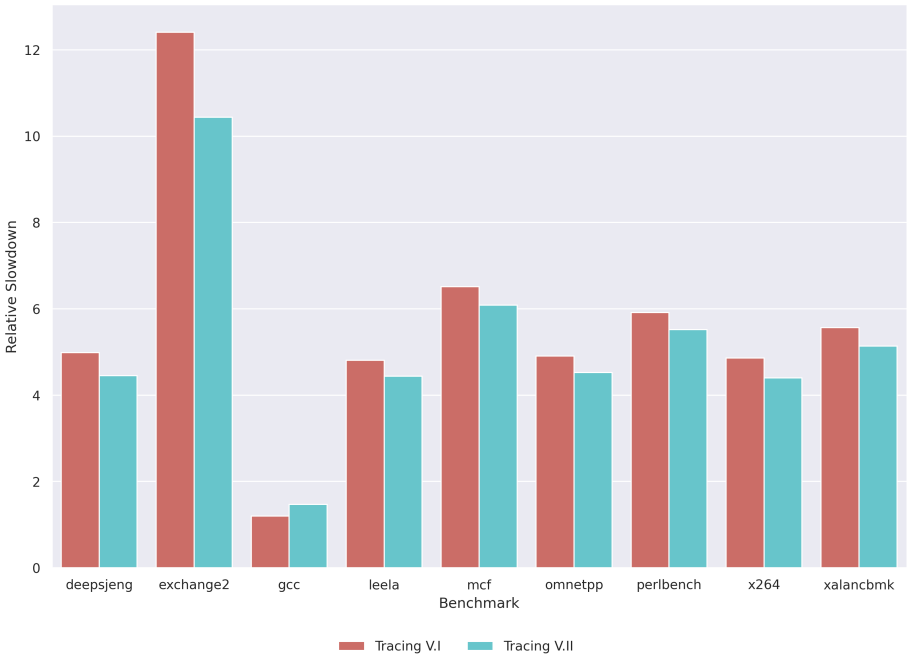


Figure 9: Relative slowdown on benchmarks from both software tracing methods

Looking at Figure 9 above shows the effects the second software tracing method had on the runtime of several benchmarks. Included in the graph are the results of the first software method allowing for a comparison between the two approaches. Analysing these results the most prominent takeaway is that overall, this approach is slightly faster than the initial software version but only slightly.

Though this approach re-enables direct chaining, the reason for not seeing a drastic improvement in performance is that to re-enable direct chaining, a call needs to be added to every single translated block. This call takes execution out of the translated code and back into QEMU code to perform the log of the basic block. This will naturally introduce a lot of overhead, slowing down execution.

The exchange2 benchmark appears to have the most noticeable improvement. This makes sense since that benchmark contains a lot of chained blocks. Since this approach allows for the performance gains of direct chaining, it explains why exchange2 is now faster than the previous software version.

On the other hand gcc is now slower. Since the gcc benchmark does not gain significant performance improvements by enabling patch chaining, by introducing an extra overhead to every translated block (via the helper call), it is now slower. To run the gcc benchmark more work is done.

3.3 Generating Intel PT Data

3.3.1 Intel PT Background

Before implementing hardware tracing methods which utilise Intel Processor Trace (Intel PT) [15], QEMU needs to be modified such that it can collect and save Intel PT data. To understand the data that is being collected, a detailed review of how Intel PT works is required.

Intel PT is an extension of the Intel Architecture that allows the collection of information about a program's execution, such as control flow. Intel PT achieves this by providing the user with a stream of highly compressed packets generated during a program's execution.

There are two big difficulties associated with using Intel PT. Firstly, though the packets generated are highly compressed, the large number of them created leads to a significant amount of data produced (hundreds of megabytes per second per core) [21]. Secondly, due to the compressed nature of these packets, their parsing is quite complex. The combination of these two features can make the parsing of this data slow, up to two to three orders of magnitude longer than the time taken to execute the original program [21].

Intel PT has several different packet types that it produces, but for the purposes of control flow tracing, only the TNT and TIP packets are significant. The TIP (Target Instruction Pointer) packet provides the current instruction pointer (IP) value. The TIP packet is generated in several different scenarios. The most common scenario is when the processor makes a computer jump.

For example, the following jump instruction: `jmp eax` causes the CPU to jump to the location stored in the `eax` register. In this situation, the resulting location of the jump cannot be determined from reading the assembly code. As such, a TIP packet is produced.

This leads to one of the key complexities in parsing Intel PT data. The trace of a program cannot be determined from the Intel PT packet stream alone; the source code being executed is needed. If a direct jump (e.g. `jmp 0x56789AB1`) is executed, Intel PT will not provide any packet information. From knowing the IP value prior to executing a direct jump, the parser can work out the result of direct jumps from reading the source code.

The use of TIP packets and source code allows a parser to reconstruct a trace in the event of computed and direct jumps, but what about conditional jumps (e.g. `je 0x56789AB1`)? This is where taken/not taken (TNT) packets come in. TNT packets provide the result of the next n conditional jumps. For example, Intel PT may produce a TNT packet with the following body 0101. This packet indicates that for the next four conditional jumps, the following will happen: first not taken, second taken, third not taken, and fourth taken.

```c
int calculate_sum(
    int limit
) {
    int sum = 0;

    for(int i = 0; i < limit; i++) {
        sum += i;
    }

    return sum;
}
```

(a) Input C Code

```
1  calculate_sum(int):
2  pushq   %rbp
3  movq    %rsp, %rbp
4  movl    %edi, -20(%rbp)
5  movl    $0, -4(%rbp)
6  movl    $0, -8(%rbp)
7  jmp     .L2
8  .L3:
9  movl    -8(%rbp), %eax
10 addl    %eax, -4(%rbp)
11 addl    $1, -8(%rbp)
12 .L2:
13 movl    -8(%rbp), %eax
14 cmpl    -20(%rbp), %eax
15 jl      .L3
16 movl    -4(%rbp), %eax
17 popq    %rbp
18 ret
```

(b) Output Assembly Code

Figure 10: Translation of a Simple Function

Figure 10, above contains the assembly code and corresponding C code seen earlier. Recall that: block 0 (B0) is lines 1-7, block 1 (B1) is lines 8 to 11, block 2 (B2) is lines 12 to 15, and block 3 (B3) is lines 16 to 18.

If the `calculate_sum()` function were called with a limit of 3, then a single TNT packet with a value of 11101 would be produced. The first three ones represent that the jump instruction on line 15 is taken. The zero represents that the jump instruction is not taken on its fourth occurrence. The final one represents that the return instruction on line 18 is taken. Though the return instruction is not a conditional jump, Intel PT still generates a TNT value for it to make parsing easier.

This information allows a parser to generate the following trace: B0, B2, B1, B2, B1, B2, B1, B2, B3. Note that throughout this entire execution, not a single TNT packet is generated, and the entire trace is represented by only 5 bits. This is the level of compression that is seen with Intel PT and its complexity.

3.3.2 Intergrating Perf Into QEMU

To start recording Intel PT data via the Linux Perf API, some setup is required. This setup is implemented in the `init_ipt()` function, shown below in Figure 11. This function is called once at the start of QEMU. To use the perf API, first, a perf event open system call is made [22]. This system call requires a `perf_event_attr` struct as input and returns a file descriptor. This struct tells the kernel what kind of perf event is being created. In this case, it is an Intel PT perf event. The value for this type of event is not fixed and needs to be read from a file hence the call to `get_intel_pt_perf_type()`.

The file descriptor returned from making this system call can be used to map data relating to the created perf event into user memory. The first thing which needs to be mapped is the header (done via the first `mmap()` call). The header allows a user to configure the Intel PT perf event and get access to the various perf buffers.

With access to the header, the buffers can be configured. In perf, there are two buffers a data buffer and an auxiliary (aux) buffer. Intel PT only writes data to the aux buffer, so the data buffer can be ignored.

To set up the aux buffer, first, its size must be set in the header. The bigger this buffer can be, the better. On the machines used for evaluation, the max possible size was 1024 pages. After setting the size attribute in the header, the aux buffer can be mapped into memory. This is done via the second `mmap()` call.

The aux buffer can be configured as either a linear buffer or a circular buffer. By mapping the memory as read and write, a linear buffer is produced. In linear buffer mode, it is easier to prevent data loss; hence this mode was chosen.

After this buffer is created, a new thread is spawned to start reading data from it. This thread executes the `intel_pt_recording_thread()` function shown below in Figure 12.

```c
// trace/ctrace.c
static void init_ipt(void)
{
    // Set-up the perf_event_attr structure
    struct perf_event_attr pea;

    memset(&pea, 0, sizeof(pea));

    pea.size = sizeof(pea);
    pea.type = get_intel_pt_perf_type();
    pea.disabled = 1; // start intel pt recording off
    pea.exclude_kernel = 1; // ignore kernal code
    pea.config = 0x2001; // default config: only interested in tracing

    // create perf file descriptor
    perf_fd = syscall(SYS_perf_event_open, &pea, 0, -1, -1, 0);

    // Map perf structures into user memory
    header = mmap(
        NULL, (NR_DATA_PAGES + 1) * PAGE_SIZE,
        PROT_READ | PROT_WRITE, MAP_SHARED,
        perf_fd, 0
    );

    header->aux_offset = header->data_offset + header->data_size;
    header->aux_size = NR_AUX_PAGES * PAGE_SIZE;

    // Create Intel PT data buffer
    aux_area = mmap(
        NULL, header->aux_size,
        PROT_READ | PROT_WRITE, MAP_SHARED,
        perf_fd, header->aux_offset
    );

    pthread_create(&trace_thread, NULL, intel_pt_recording_thread, NULL);
}
```

Figure 11: Intel PT Perf Setup

Since the created aux buffer is in linear mode, data is written to it at the beginning, and it fills up with data. Once the buffer is full, it wraps around, and data start to be written from the start again. Note it will override data if the user does not read from it fast enough.

To get access to the current top of the buffer, the header has an aux_head attribute. This value is updated by the kernel when new data comes in. After reading the data, the user must update the headers aux_tail attribute. This lets the Kernal know where the user has gotten to in the buffer, so it can safely overwrite old data. If the kernel has to override unread data, it will set a flag, so the user is aware of this.

The `intel_pt_recording_thread()` function described earlier implements the process of reading this data. Seen below in Figure 12, this function, when called, continuously reads the current aux head value. If that value has increased since it was last read, then there is new data which can be saved. The function saves this new data to an Intel PT data file which can be used by a parser later. After writing the data, it updates the aux_tail attribute as required.

To ensure this function can exit, if the head value does not change between reads, it checks if a can_stop_recoring boolean has been set. If this boolean has been set, the loop breaks, causing the tracing thread to stop. This boolean is set to true in the exit code of QEMU, ensuring that the thread doesn't go on forever.

```
// trace/ctrace.c
static void* intel_pt_recording_thread(void)
{
    const unsigned char *buffer = (const unsigned char *)aux_area;
    u64 size = header->aux_size;
    u64 last_head = 0;

    while(true) {
        u64 head = header->aux_head;

        if (head == last_head) {
            if(can_stop_recoring) break;
            else continue;
        }

        u64 wrapped_head = head % size;
        u64 wrapped_tail = last_head % size;

        if(wrapped_head > wrapped_tail) {
            // fwrite from tail --> head
            fwrite(/* ... */);
        } else {
            // fwrite from tail -> size
            fwrite(/* ... */);

            // fwrite start --> head
            fwrite(/* ... */);
        }

        last_head = head;
        header->aux_tail = head;
    }

    return NULL;
}
```

Figure 12: Intel PT Recording Thread Setup

The system for recording Intel PT data via perf is now complete. All that is left is to create functions that allow QEMU to tell perf when to start and stop generating the Intel PT data. By default, perf will not generate Intel PT data and needs to be told when to start/stop.

To start/stop recording, a enable/disable system call is made to the perf file descriptor created in the previous steps. This system call is wrapped by the `intel_pt_recording_start()`, and `intel_pt_recording_stop()` functions for ease of use.

3.4 Hardware Version I

With the functions `intel_pt_recording_start()` and `intel_pt_recording_stop()`, which start and stop recording Intel PT data implemented. The next step is to design and implement a tracing method which uses the data these functions produce.

The first design decision to make is, when should the Intel PT recording start and stop. One option would be to start recording Intel PT data when QEMU starts and stop when it exits. The problem with this approach is that it will lead to a large amount of waste.

In order to generate a trace of the guest program, only the execution of translated code needs to be recorded, not all of QEMU. To record this information, the `cpu_tb_exec()` function can be modified to start/stop Intel PT data recording when entering/exiting translated code. The result of this modification is shown in Figure 13.

```
// cpu-exec.c
static inline TranslationBlock * QEMU_DISABLE_CFI
cpu_tb_exec(CPUState *cpu, TranslationBlock *itb, int *tb_exit)
{
    CPUArchState *env = cpu->env_ptr;
    uintptr_t ret;
    TranslationBlock *last_tb;
    const void *tb_ptr = itb->tc.ptr;

    // ...

    intel_pt_recording_start();
    ret = tcg_qemu_tb_exec(env, tb_ptr);
    intel_pt_recording_stop();

    // ...

    return last_tb;
}
```

Figure 13: cpu_tb_exec source code

There is one problem with the modifications made to the `cpu_tb_exec()` function. That is that the function `tcg_qemu_tb_exec()` doesn't always return after being called. Meaning that `intel_pt_recording_stop()` will not be called, and the Intel PT data recording will never stop.

In the event that an exception is raised by translated code, QEMU will jump to an exception helper. To ensure that Intel PT data recording is always stopped, the exception code needs to be modified such that it will stop Intel PT recording. The result of this can be seen in Figure 14.

```
// arm/os_helper.c
void HELPER(exception_with_syndrome)(
        CPUARMState *env, uint32_t excp,
        uint32_t syndrome
) {
    intel_pt_recording_stop();
    raise_exception(env, excp, syndrome, exception_target_el(env));
}
```

Figure 14: exception_with_syndrome source code

With the system for recording Intel PT data developed, the next step is to deal with the fact that an Intel PT parser needs to know the assembly code being executed (so that it can understand TNT packets). Whilst this could normally be obtained by disassembling the source code being executed, the code running in QEMU is generated at run time and cannot be determined by reading the QEMU binary.

It could be possible to get this assembly code by modifying the QEMU translation phase, but this is quite complex, so it would be preferable to be able to generate a trace without needing the assembly code. Secondly, if the parser did not need assembly code information, it would be simpler to implement.

This approach is possible, but it requires a deep understanding of how QEMU executes translated code. Figure 13 above shows the function which executes translated code is `tcg_qemu_tb_exec()`. This function takes in a pointer to the CPU Struct and a pointer to the start of the translated code to execute.

To understand how QEMU executes code, an understanding of this function is required. There is a problem with this; the function `tcg_qemu_tb_exec` does not exist. `tcg_qemu_tb_exec` is not actually a function defined within QEMU, but it is a pointer to code which is generated at runtime by QEMU.

Thankfully this function is always the same, and QEMU can output it for debugging purposes, shown below in Figure 15. Note this debugging method cannot be used to generate assembly code for an Intel PT parser because it slows down execution significantly, far slower than software tracing methods.

```
PROLOGUE:
0x7f7910000000:  pushq  %rbp
0x7f7910000001:  pushq  %rbx
0x7f7910000002:  pushq  %r12
0x7f7910000004:  pushq  %r13
0x7f7910000006:  pushq  %r14
0x7f7910000008:  pushq  %r15
0x7f791000000a:  movq   %rdi, %rbp
0x7f791000000d:  addq   $-0x488, %rsp
0x7f7910000014:  jmpq   *%rsi
0x7f7910000016:  xorl   %eax, %eax
0x7f7910000018:  addq   $0x488, %rsp
0x7f791000001f:  vzeroupper
0x7f7910000022:  popq   %r15
0x7f7910000024:  popq   %r14
0x7f7910000026:  popq   %r13
0x7f7910000028:  popq   %r12
0x7f791000002a:  popq   %rbx
0x7f791000002b:  popq   %rbp
0x7f791000002c:  retq
```

Figure 15: tcg_qemu_tb_exec asm code

There's a lot to unpack with the asm code in Figure 15. One important piece of information is that the `rdi` and `rsi` are the first and second integer arguments supplied to this function when calling it [16]. Since `tcg_qemu_tb_exec()` is called like so `tcg_qemu_tb_exec(env, tb_ptr);` it means that the `rdi` register points to the CPU Struct and the `rsi` register points to start of the translated code.

The fact that the `rsi` register points to translated code means that the `jmpq` instruction at position 0x7f7910000014 is where QEMU jumps to and thus starts executing translated code. Since this is a computed jump (meaning the destination is not known from looking at the assembly code), a TIP packet will be produced. Using this information, a parser can determine the address of translated blocks, which are executed by parsing the TIP packets produced.

However, there is still the problem of direct chaining. Recall that in the computed chaining method, the translated block will make a call to the `lookup_tb_ptr()` helper function. This helper function returns an address that points to either: another translated block of code, or the QEMU epilogue code. In either of these cases, the translated block which made this call will be making a computer jump to the address returned by this function. This computed jump means a TIP packet is produced. As such, this form of direct chaining does not need to be modified.

This leaves the patch chaining method. This method is more complex and requires analysing translated assembly to understand.

```
IN:
0x5500818520:  hint  #0x22
0x5500818524:  mov  x0, sp
0x5500818528:  bl  #0x5500818f04
```

(a) Input Guest Code

```
OUT:
  -- guest addr 0x0000005500818520
0x7...100:  movl  -8(%rbp), %ebx
0x7...103:  testl  %ebx, %ebx
0x7...105:  jl  0x7f3b9b3ff149
  -- guest addr 0x0000005500818524
0x7...10b:  movq  0x138(%rbp), %rbx
0x7...112:  movq  %rbx, 0x40(%rbp)
  -- guest addr 0x0000005500818528
0x7...116:  movabsq  $0x550081852c, %rbx
0x7...120:  movq  %rbx, 0x130(%rbp)
0x7...127:  jmp  0x7f3b9b3ff12c
0x7...12c:  movabsq  $0x5500818f04, %rbx
0x7...136:  movq  %rbx, 0x140(%rbp)
0x7...13d:  leaq  -0x104(%rip), %rax
0x7...144:  jmp  0x7f3b9b3ff018
```

(b) Before goto tb patching

```
OUT:
  -- guest addr 0x0000005500818520
0x7...100:  movl  -8(%rbp), %ebx
0x7...103:  testl  %ebx, %ebx
0x7...105:  jl  0x7f3b9b3ff149
  -- guest addr 0x0000005500818524
0x7...10b:  movq  0x138(%rbp), %rbx
0x7...112:  movq  %rbx, 0x40(%rbp)
  -- guest addr 0x0000005500818528
0x7...116:  movabsq  $0x550081852c, %rbx
0x7...120:  movq  %rbx, 0x130(%rbp)
0x7...127:  jmp  0x7f3b9b3ff240
0x7...12c:  movabsq  $0x5500818f04, %rbx
0x7...136:  movq  %rbx, 0x140(%rbp)
0x7...13d:  leaq  -0x104(%rip), %rax
0x7...144:  jmp  0x7f3b9b3ff018
```

(c) After goto tb patching

Figure 16: Translated Block assembly code

To demonstrate how the patch chaining method works, there is an example of it included in Figure 16 above. Figure 16a contains the input guest assembly of a basic block (block 0). This block ends with a jump to another guest block (block 1). Figure 16b contains the initial translated target code for block 0, and Figure 16c contains the translated code after it has been patched as a result of the patch chaining method.

Looking at the initial translation of block 0 (Figure 16b), the instructions from address 0x7f3b9b3ff12c to the end of the block handle exiting the block and returning to QEMU. These instructions insert the guest address of block 1 (0x5500818f04), into the CPU struct and return to QEMU. QEMU reads this guest address and finds the corresponding translation block (or translates it if one doesn't exist), and begins executing it.

An interesting item to note about the translated assembly code in Figure 16b is the direct jump at address 0x7f3b9b3ff127. This jump performs a direct jump to the next instruction. Doing so appears entirely pointless. If it weren't included, nothing would change. However, when looking at the same jump in Section 16c, its address has changed.

This is how QEMU implements direct chaining. When it translates block 0 initially, it recognises the direct jump at the end of the block. However, if that block hasn't been translated, it does not know what target address to jump to. Instead, QEMU includes the instructions from address 0x7f3b9b3ff12c to the end of the block. These instructions return to QEMU so a lookup can be performed.

It would be beneficial for performance if returning to QEMU could be avoided, and instead, block 0 performs a direct jump to block 1. To allow for this QEMU also includes the jump at address 0x7f3b9b3ff127. When the translated code for block 0 returns to QEMU for the first time, it will translate block 1. By translating block 1, QEMU has access to the address of this block. It can update or 'patch' the jump in block 0 at location 0x7f3b9b3ff127 to be a direct jump to the translated code for block 1. Then if block 0 is ever executed again, it will not return to QEMU. Instead, it will jump directly to block 1 and begin executing that.

Having analysed how the 'goto tb + exit tb' direct chaining method works, it is clear that the design of this hardware tracer will not work; when patch chaining is enabled, direct jumps are used to go from one block to another. As such, no TIP packet will get produced. For the parser to be able to follow execution through this jump, it will need access to the assembly code being executed. To keep the initial hardware parser simple, patch chaining will need to be disabled.

More information is needed for the parser to be able to generate a complete trace. Currently, the parser will be able to generate a trace of translated blocks. A trace of the original guest basic blocks is wanted, not their translations.

To achieve this, the parser needs to be able to map from a translated block address to its original guest block address. If the parser can perform this mapping, whenever it sees a TIP packet, it can check if it has a mapping to a guest block. If the mapping exists, the guest address can be added to the trace.

To get the desired information, the QEMU translation function is modified to log the guest address of each basic block and their corresponding translated block's address. The result of this can be seen in Figure 17 below.

```
// arm/translate.c
/* generate intermediate code for basic block 'tb'.  */
void gen_intermediate_code(
        CPUState *cpu, TranslationBlock *tb, int max_insns,
        target_ulong pc, void *host_pc
) {
    DisasContext dc = { };
    const TranslatorOps *ops = &arm_translator_ops;
    CPUARMTBFlags tb_flags = arm_tbflags_from_tb(tb);

    // ...

    translator_loop(cpu, tb, max_insns, pc, host_pc, ops, &dc.base);
    record_host_to_guest_mapping(tb->pc, tb->tc.ptr);
}
```

Figure 17: gen_intermediate_code source code source code

The current system will work for small and some large programs. However, it will not work for all programs. There is a problem with the method of recording Intel PT data created in the previous section. There is a chance that the Intel PT buffers will fill up, causing data to be lost. If this were to occur, a complete trace could not be generated.

During testing, it was discovered that the perf interface does not update its head and tail pointers when recording Intel PT data. Meaning if the execution of translated code goes on for too long without returning to QEMU, which causes the Intel PT recording to stop, eventually, the buffers will fill up.

A possible solution to this problem is to disable all forms of direct chaining, forcing QEMU to only execute a single translated block at a time. Another solution is to modify the `helper_log_guest_block_exec()` (computed chaining) function so that it only allows QEMU to execute a fixed number of translated blocks, after which it forces execution to return to QEMU.

The second solution is preferable for two reasons. Firstly even if only a limited number of blocks can be executed via computed chaining at a time, it will still have performance benefits over always returning to QEMU. Secondly, in future, both methods of direct chaining will be wanted, so if this problem can be solved now, it will make the implementations of later methods easier.

To implement this feature, a chain count attribute is added to the QEMU CPU struct. This chain count is set to a fixed amount before executing a translated block. It is decremented in the `helper_log_guest_block_exec()` function every time it is called. If it ever reaches zero, the function forces the translated code to return to QEMU. This allows a limit to be placed on the number of translated blocks which can be executed before stopping, causing the Intel PT data recording to be stopped. The result of this modification is shown below in Figure 18.

```
// cpu-exec.c
const void *HELPER(lookup_tb_ptr)(CPUArchState *env)
{
    // ...

    if (env->chain_count-- == 0) {
        return tcg_code_gen_epilogue;
    }

    // ...
}

static inline TranslationBlock * QEMU_DISABLE_CFI
cpu_tb_exec(CPUState *cpu, TranslationBlock *itb, int *tb_exit)
{
    // ...
    env->chain_count = CHAIN_COUNT_START_AMOUNT;

    intel_pt_recording_start();
    ret = tcg_qemu_tb_exec(env, tb_ptr);
    intel_pt_recording_stop();

    // ...
}
```

Figure 18: lookup_tb_ptr and cpu_tb_exec code

The problem of the buffers being filled up from code execution going on for too long has been solved. However, the buffers could still get full if QEMU is executing code too quickly, regardless of it stopping and then starting again.

Before QEMU starts execution, it needs to check if the buffers are nearing their max capacity. If so, wait before starting execution. To do this, the `intel_pt_recording_start()` function and the `intel_pt_recording_thread()` function created in the previous section need to be modified.

The `intel_pt_recording_start()` function is modified to include a spin lock which waits for the `intel_pt_recording_thread()` function to signal when there is enough space in the buffers for execution to continue.

To keep the implementation simple, whenever the recording thread is copying data, it will force the spin lock to wait. There is the possibility that the spin lock will miss the lock being set. As long as the chain count is configured such that the buffers can only get 50% full, this will not be an issue. It is not possible for the spin lock to miss the lock being set more than once. The results of these modifications are shown below in Figure 19.

```
// trace/ctrace.c
static inline void
intel_pt_recording_start(void)
{
    while (pt_thread_recording) {}
    // ...
}

static void
intel_pt_recording_thread()
{
    // ...

    while(true) {
        // ...

        pt_thread_recording = true;

        // ...

        pt_thread_recording = false;
    }
}
```

Figure 19: Spin Lock Code

This hardware tracer is much more complex than the previous software approaches. In summary, it works by doing the following. A separate thread has been added to QEMU. This thread is constantly waiting for Intel PT data that it receives through the Linux perf interface. Whenever QEMU starts executing translated code, it starts Intel PT data recording. When the execution finishes, it stops Intel PT data recording.

To ensure that the buffers do not overflow, QEMU has been modified to exit out of translated code after executing a fixed number of translated blocks. QEMU will also wait for the buffers to be emptied before starting Intel PT data recording.

To ensure that the parser has sufficient information to turn the Intel PT data into a valid trace, a mapping from target address to guest address is outputted. This mapping contains an entry for each basic block. Lastly, the patch chaining has been disabled as the parser does not have sufficient information to follow that form of direct chaining.

Having successfully implemented the first hardware tracing method, as with all previous methods, a small evaluation is performed. The same benchmarks as before were used, the results of which can be seen below in Figure 20.

Looking at these results, they show that this method appears to be slightly faster than the second software tracing method. However, as expected by disabling patch chaining the performance is comparable to software tracing.

Lastly, these results only take into account QEMU execution time, not the time to parse that data into the final trace. Parsing time will likely cause this method to be slower overall when compared with software. As such, the hardware method needs to be improved.

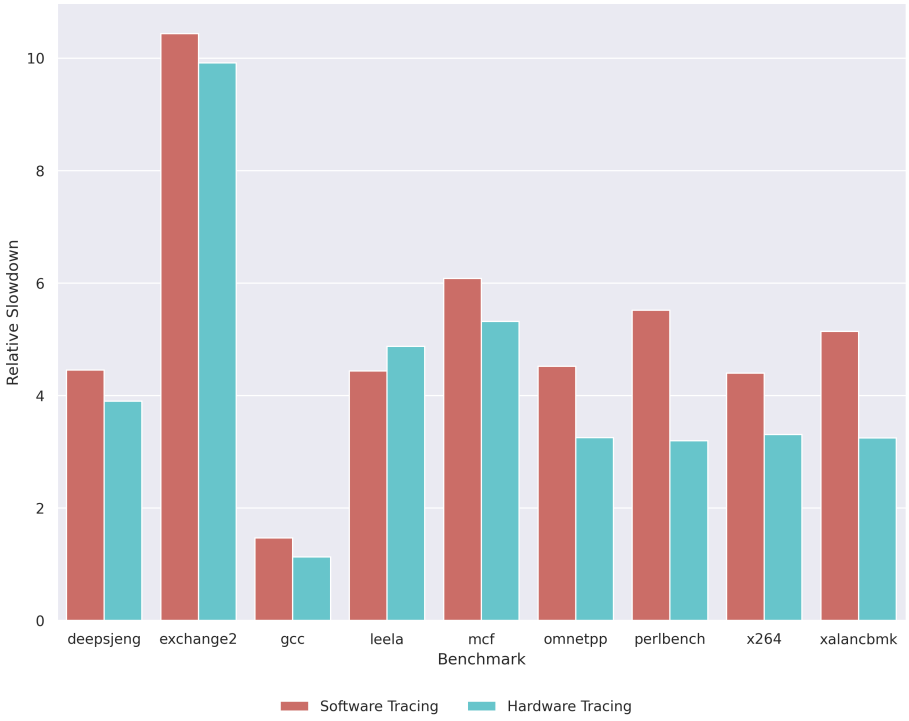


Figure 20: Relative slowdown on benchmarks from hardware tracing method

3.5 Hardware Version II

As expected the initial hardware version is slow due to direct chaining needing to be disabled. To overcome this problem it will need to be enabled, but as discussed before that requires the parser to have access to the assembly code being executed.

To gain this assembly information the QEMU translation phase needs to be modified. To improve performance there is no need to record all assembly code, only instructions which change control flow are needed to be known: these are jumps and call instructions.

The high-level process of getting this information is to populate an asm log file with the following information:

- When QEMU starts translating a new block log a new block identifier with the start address of this new block
- Every time QEMU translates a jump instruction log it to the file, indicating if it is a conditional or direct jump
- Every time QEMU translates a function call log it to the file, indicating the function which is being called
- When QEMU finishes translating the block log its final size

There are a few complications with this process, for some jumps the location is a label. The address of labels is not known at the translation time of the jump instruction. However, the label ID can be logged as this jumps location and when the label is translated it can be logged along with its current location. This can allow the parser to produce a mapping from label ID to address.

A final complication is that due to patch chaining, a jump's target location can change. To give the parser this information whenever a patch occurs an update needs to be logged to the asm file. This update indicates which jump has changed and what its new target location is.

A summary of the modifications implemented within QEMU to provide this information can be seen below in Figure 21.

```
// tcg/translate.c
/* Called with mmap_lock held for user mode emulation.  */
TranslationBlock *tb_gen_code(
        CPUState *cpu, target_ulong pc,
        target_ulong cs_base, uint32_t flags, int cflags
) {
    CPUArchState *env = cpu->env_ptr;
    TranslationBlock *tb, *existing_tb;

    // ...
    fprintf(asm_log_file, "BLOCK: 0x%lX\n", tb->tc.ptr);

    // ...
    gen_intermediate_code(cpu, tb, max_insns, pc, host_pc);
    // ...

    fprintf(asm_log_file, "BLOCK_SIZE: %lu\n", tb->tc.size);
    // ...
}
// tcg/tcg.c
static void tcg_reg_alloc_call(TCGContext *s, TCGOp *op ) {
    // ...
    fprintf(
        asm_log_file , "CALL: 0x%lX %s\n",
        s->code_ptr, info->name
    );
    // ...
}
// tcg/i386/tcg-target.inc.c
static void tcg_out_jxx(
        TCGContext *s, int opc, TCGLabel *l, int small
) {
    //...
    fprintf(asm_log_file, "JXX: 0x%lX %u\n",  l->id);
}
static void tcg_out_jmp(
        TCGContext *s, const tcg_insn_unit *dest
) {
    // ...
    fprintf(
        asm_log_file, "JMP: 0x%lX 0x%lX\n",
        jump_loc, dest
    );
}
```

Figure 21: asm logging code

The parser now has access to the translated assembly. However, having access to the assembly code is not enough information. The assembly code being executed is generated throughout the runtime of QEMU, not all at once, and can change throughout the runtime.

This means the parser cannot read in all the assembly at once. The parser needs to have a method of keeping its current assembly code representation in line with where the program is in execution. This can be achieved by utilising the fact that translation happens outside of translated code execution.

If a marker is sent to both the asm log file and the Intel PT data file when recording starts, this will allow the parser to keep its assembly code representation inline with the true value. When the parser sees this marker in the Intel PT data, it can parse up to the next marker in the asm log file.

As the asm log file only contains text data adding a marker is straightforward. Whenever the Intel PT start recording function is called, it can write the marker to the asm log file.

Adding a marker to the Intel PT data file is more complex. A possible solution would have been to utilise the PT Write feature in Intel PT. This allows a program to write data in the form of a PTW packet to the Intel PT packet stream. However, this feature was not available on the CPU used for development.

The chosen solution was to use the fact that Intel PT is always started in the same location. When Intel PT starts recording, it sends a TIP packet indicating the current address. Since this is always the same, it can be used as a marker for Intel PT recording starting. The parser can determine this address as it will be the first TIP packet it sees. Thus, whenever the parser sees this address again in future, it knows it needs to advance to the next marker in the asm log.

To demonstrate, this system, Figure 22, below, includes the true assembly code of a translated block with that same block as seen within the asm log file.

```
0x7...100:  movl     -8(%rbp), %ebx
0x7...103:  testl    %ebx, %ebx
0x7...105:  jl       0x7f56077ff15d
0x7...10b:  movq     0x138(%rbp), %rbx
0x7...112:  movq     %rbx, 0x40(%rbp)
0x7...116:  movabsq  $0x550081852c, %rbx
0x7...120:  movq     %rbx, 0x130(%rbp)
0x7...127:  decl     0x13284(%rbp)
0x7...12d:  cmpl     $0, 0x13284(%rbp)
0x7...134:  je       0x7f56077ff140
0x7...13a:  nop
0x7...13b:  jmp      0x7f56077ff140
0x7...140:  movabsq  $0x5500818f04, %rbx
0x7...14a:  movq     %rbx, 0x140(%rbp)
0x7...151:  leaq     -0x118(%rip), %rax
0x7...158:  jmp      0x7f56077ff018
0x7...15d:  leaq     -0x121(%rip), %rax
0x7...164:  jmp      0x7f56077ff018
```

(a) True Assembly Code

```
BLOCK: 0x7F56077FF100
JXX: 0x7F56077FF105 0
JMP: 0x7F56077FF13B 0x7F56077FF140
JXX: 0x7F56077FF134 0x7F56077FF140
JMP: 0x7F56077FF158 0x7F56077FF018
LBL: 0 7F56077FF15D
JMP: 0x7F56077FF164 0x7F56077FF018
BLOCK_SIZE: 105
```

(b) Log of Assembly Code

Figure 22: Translated Block Assembly Code Log Representation

With the problem of keeping the parser's representation translated assembly code in line with the truth solved, that leaves one final problem. That problem is helper calls. When a translated block makes a call to a QEMU helper function, it jumps to a function in the QEMU source code.

In theory, executing helper functions shouldn't be a problem. However, the current system does not have a method of knowing what assembly code is being executed, and this is needed. The parser needs to follow execution into these helper calls so it can follow it out of the helper calls back into the translated block. There are three possible solutions to this problem.

The first solution is to disable return compression in Intel PT. Intel PT compresses return calls. This compression means that the parser needs to have its own call stack. The parser can use the call stack to determine the address to jump to when executing a return instruction.

Intel PT does have the option to disable return compression [21], allowing the parser to ignore calls and wait for a TIP packet indicating the call has finished. However, attempts to disable return compression did not work, and thus this option could not be used.

A second option would be to decompile the entire QEMU binary and feed that into the parser. The parser would then have all the needed assembly code to follow helper calls. However, the QEMU codebase is large and parsing that much assembly code would add a significant overhead to trace generation.

Another problem with decompiling QEMU is that if a QEMU function calls a standard library function, the assembly of that would also be needed. However, standard library functions are not contained in the QEMU codebase, so all dynamically linked libraries would also need to be decompiled.

The third and final option is to disable Intel PT when entering helper calls and re-enable it when returning to translated code. When Intel PT recording starts, it forces a TIP packet to be generated. This TIP packet will allow the parser to skip tracing the QEMU source code and only follow translated code. A possible benefit of this method is that it reduces the size of the Intel PT data being parsed, as there will be no Intel PT data generatedd for helper calls. Since this approach is the simplest and most feasible method, it is the one used for this tracing implementation.

To disable Intel PT recording when executing helper calls, two additional helper functions were created. The first disables the Intel PT recording when called, and the second re-enables the Intel PT recording.

QEMU can be modified to do the following whenever generating a helper call. First, generate a call to the stop Intel PT recording helper. Then generate the desired helper call. Lastly, generate a call to the start Intel PT recording helper. The code to do so is shown below in Figure 23.

The code in this figure works by placing a wrapper over the original call generator. Now, whenever the function to generate a call is called, it will generate three calls. The first stops Intel PT recording, the second is the given call to generate, and the third starts Intel PT recording.

```
// tcg/tcg.c
static inline void _tcg_gen_callN(
        void *func, TCGTemp *ret, int nargs, TCGTemp **args
) {
    // ...
}

void tcg_gen_callN(
        void *func, TCGTemp *ret, int nargs, TCGTemp **args
) {
    _tcg_gen_callN(helper_stop_intel_pt, dh_retvar(void), 0, NULL);
    _tcg_gen_callN(func, ret, nargs, args);
    _tcg_gen_callN(helper_start_intel_pt, dh_retvar(void), 0, NULL);
}
```

Figure 23: New QEMU Helper Call Generator

This hardware tracing method is now complete and will work for small and some large programs, but as with the previous implementation, there is the same problem with the buffers filling up.

Recall that in the previous implementation, a chain count was introduced to solve this problem. This chain count forces QEMU to exit out of translated code, if it had been going for too long. However, this chain count is only implemented for the computed chaining method. It needs to be modified to work for patch chaining.

The chain count introduced in the previous method was added as an attribute to the QEMU CPU struct. Recall that this CPU Struct is passed into the `tcg_qemu_tb_exec()` function via the `rsi` register, the value of this register is then moved into the `rdi` register. This means that all translated blocks have access to the CPU struct. So, to implement the chain count into the patch chaining method assembly code needs to be added to each basic block to utilise this chain count.

Recall that the patch chaining method described earlier works by modifying a direct jump at the end of a translated block of code. To bring the chain count into this method prior to the jump executing the following needs to happen: The chain count needs to be decremented by one; then the chain count is compared with zero; if it is equal to zero the translated block should return to QEMU; if the chain count is not zero then the normal direct chaining method can continue.

Figure 24 shows the result of adding this assembly to each translated block. Figure 24b is the original translation of a guest block. Figure 24c contains the same translation with the new assembly code added.

The new code placed at location 0x7ff7d0000127 decrements the chain count by 1. Then the chain count is compared with zero. Next, if the chain count is zero, it jumps to the first instruction after the patch chaining jump. This will return execution back to the main QEMU loop. If the chain count is not zero, then the patch chaining jump is executed. This jump will either exit back to QEMU or go to another translated block if patching has occurred.

```
IN:
0x5500818520:  hint  #0x22
0x5500818524:  mov  x0, sp
0x5500818528:  bl  #0x5500818f04
```

(a) Input Guest Code

```
OUT:
  -- guest addr 0x0000005500818520
0x7...100:  movl  -8(%rbp), %ebx
0x7...103:  testl  %ebx, %ebx
0x7...105:  jl  0x7ff7d0000149
  -- guest addr 0x0000005500818524
0x7...10b:  movq  0x138(%rbp), %rbx
0x7...112:  movq  %rbx, 0x40(%rbp)
  -- guest addr 0x0000005500818528
0x7...116:  movabsq  $0x550081852c, %rbx
0x7...120:  movq  %rbx, 0x130(%rbp)

0x7...127:  jmp  0x7ff7d000012c
0x7...12c:  movabsq  $0x5500818f04, %rbx
0x7...136:  movq  %rbx, 0x140(%rbp)
0x7...13d:  leaq  -0x104(%rip), %rax
0x7...144:  jmp  0x7f3b9b3ff018
```

(b) Without Chain Count

```
OUT:
  -- guest addr 0x0000005500818520
0x7...100:  movl  -8(%rbp), %ebx
0x7...103:  testl  %ebx, %ebx
0x7...105:  jl  0x7ff7d000015d
  -- guest addr 0x0000005500818524
0x7...10b:  movq  0x138(%rbp), %rbx
0x7...112:  movq  %rbx, 0x40(%rbp)
  -- guest addr 0x0000005500818528
0x7...116:  movabsq  $0x550081852c, %rbx
0x7...120:  movq  %rbx, 0x130(%rbp)
0x7...127:  decl  0x13284(%rbp)
0x7...12d:  cmpl  $0, 0x13284(%rbp)
0x7...134:  je  0x7ff7d0000140
0x7...13a:  nop
0x7...13b:  jmp  0x7ff7d0000140
0x7...140:  movabsq  $0x5500818f04, %rbx
0x7...14a:  movq  %rbx, 0x140(%rbp)
0x7...151:  leaq  -0x118(%rip), %rax
0x7...158:  ejmp  0x7ff7d0000018
```

(c) With Chain Count

Figure 24: Translated Block assembly code

As with the previous hardware method, this new hardware method is also very complex. In summary, it does the following: When translating assembly code log the key instructions (jumps and calls) to an asm log file; When jump instructions get patched log the new location to this asm file; To allow the parser to know when this asm it generated log a breakpoint indicator to this asm log prior to Intel PT recording starting; Intel PT recording always produces the same TIP packet, the Intel PT parser can use this packet to know how many asm breakpoints it should have parsed; Lastly, to deal with call instructions Intel PT recording stops and then restarts after executing them.

After successfully implementing the second hardware tracing method, as with all previous methods, a small evaluation is performed. Doing so will provide insight into what worked well and what didn't. These insights can be used to influence the next tracing methods design. The same benchmarks as before were used, the results of which can be seen below in Figure 25.

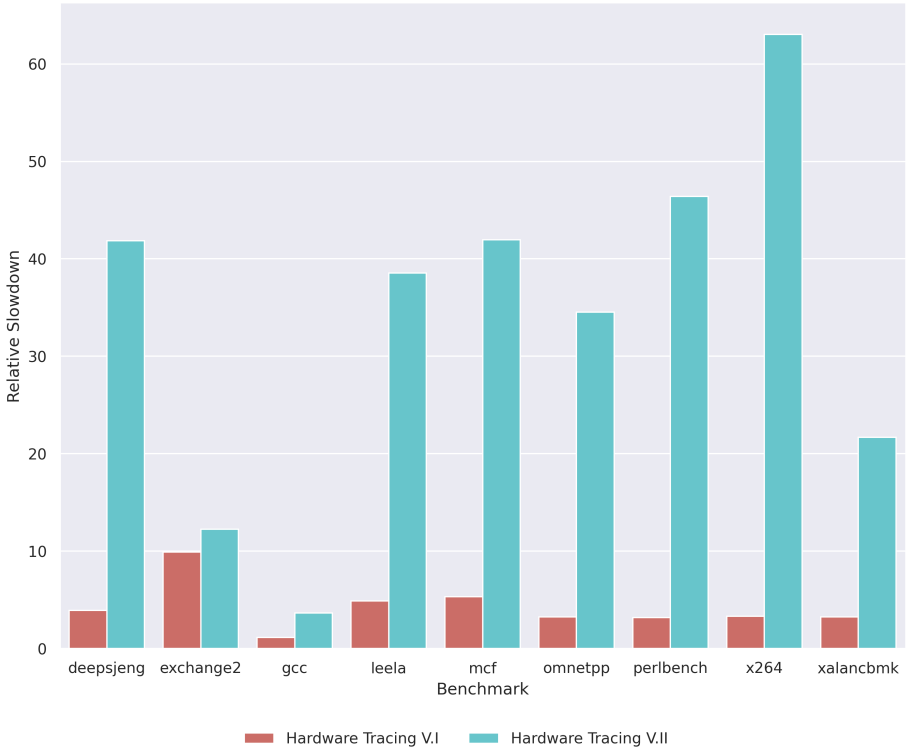


Figure 25: Relative slowdown on benchmarks from 2nd hardware tracing method

Looking at Figure 25, it is extremely apparent that this method is significantly worse than the previous one. This is somewhat surprising as direct chaining was fully enabled in this method. It is expected that execution would be the same if not faster, definitely not significantly slower.

There are two possible reasons for this slowdown. The first is that the chain count code added to basic blocks could be slowing down execution. The second possibility is that the helper calls to turn Intel PT on and off are slowing down execution. It is unlikely that the first reason is the cause of the slowdown. Though the added instructions will induce a performance hit, the gains of re-enabling direct chaining should outway that negation.

The added helper calls are likely producing this significant slowdown in performance. If it is these helper calls which are slowing down execution, it is important to determine if it is the call to the helper or the turning on and off Intel PT which is causing the slowdown.

If it is just the extra calls to helpers which is causing a slowdown, then a possible solution would be to manually inject the code to turn Intel pt on and off into all helpers. If it is the act of turning on add of Intel PT which is slowing down execution, then a different approach is required.

To find an answer to this question, the benchmarks were run again, but this time the helper calls did nothing. This will not work as a tracing method, but it will show what is causing the slowdown. The results of these benchmarks are shown in Figure 26.

Looking at the results, it is clear that it is not the helper calls which are causing the slowdown. It is the act of turning on and off Intel PT. This result does make sense. To turn on/off Intel PT recording, a system call must be made. System calls can be quite expensive and will cause a performance hit. This method adds two system calls to every helper call in QEMU, so naturally will be slow for most programs.

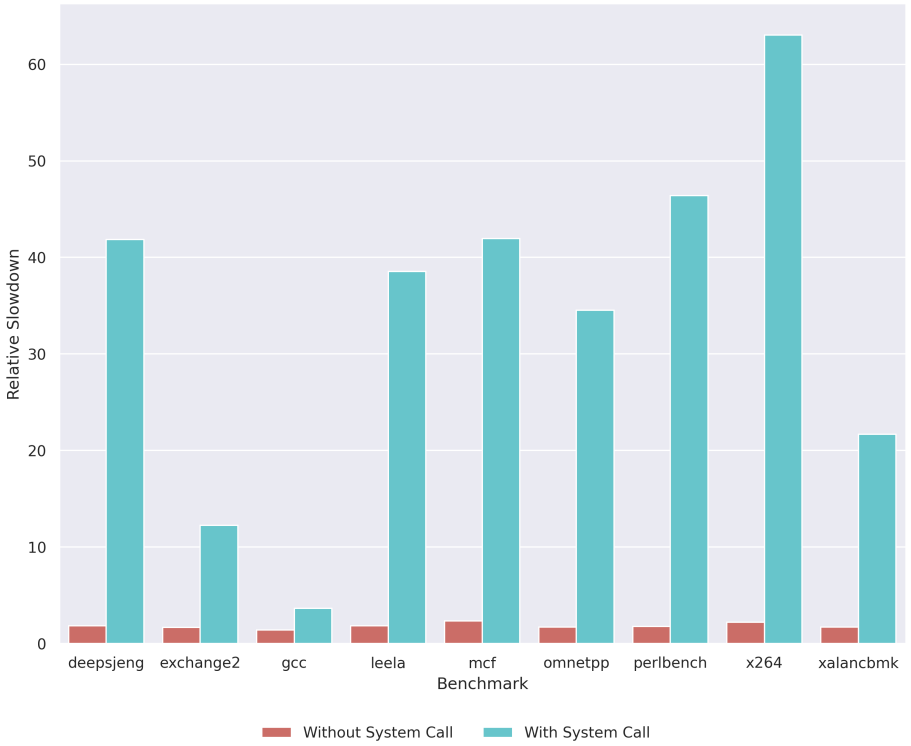


Figure 26: Relative slowdown on benchmarks from 2nd hardware tracing method with empty helper calls

3.6 Hardware Version III

From the discoveries made in the previous section, to improve the performance of the hardware tracer, Intel PT must not stop recording when entering helper calls. However, the problem of not having access to the assembly code being executed still exists.

As discussed, decompiling all of QEMU would be too much of an overhead to make using Intel PT worth it. However, it was discovered that making extra helper calls does not impede performance significantly. A possible solution is to utilise helper functions that do not affect Intel PT data recording.

Recall Intel PT generates a TIP packet every time a call to a QEMU helper function is made. This allows the parser to know when a translated block enters a helper call. The problem is without the assembly it will now know when that call has finished. What if a special helper function is created with known assembly code, then the parser could determine when this call has finished.

If a dummy helper function is created that immediately returns. The parser knows that if it sees a call to this function (via a TIP packet), next it will see a TNT packet (Intel PT marks executing return instructions via a taken value in a TNT packet), after which it means that execution has returned back to translated code.

A call to this dummy helper function could be inserted after every regular helper call. Now when a parser sees a call instruction it can wait until it sees the call to the dummy helper function. When the parser sees the dummy call it knows execution has returned from the first helper call back to the translated block, then into the dummy helper call. It can wait for the next TNT packet, after which it knows execution has returned to the translated block.

The only problem with this approach is that the parser needs to know the address of the dummy helper function. This address is what it needs to look for in the TIP packets. To provide the parser with this address, QEMU can be modified so the first call instruction generated calls the dummy helper.
Now, when the parser sees the first call instruction it knows the next TIP packet represents the address of the dummy helper function. It can use this address for tracking all subsequent helper calls.

The modifications required to implement this approach are straightforward, shown below in Figure 27.

```
// tcg/tcg.c
static inline void _tcg_gen_callN(
        void *func, TCGTemp *ret, int nargs, TCGTemp **args
) {
    // ...
}

void tcg_gen_callN(
        void *func, TCGTemp *ret, int nargs, TCGTemp **args
) {
    if (is_first_helper_generated) {
        _tcg_gen_callN(helper_dummy_call, dh_retvar(void), 0, NULL);
        is_first_helper_generated = false;
    }

    _tcg_gen_callN(func, ret, nargs, args);
    _tcg_gen_callN(helper_dummy_call, dh_retvar(void), 0, NULL);
}
```

Figure 27: QEMU Helper Call Generator

With these modifications complete the benchmarks can be re-run to see how performance has changed with the removal of turning on and off the Intel PT recording. The results of these benchmarks can be seen below in Figure 28.

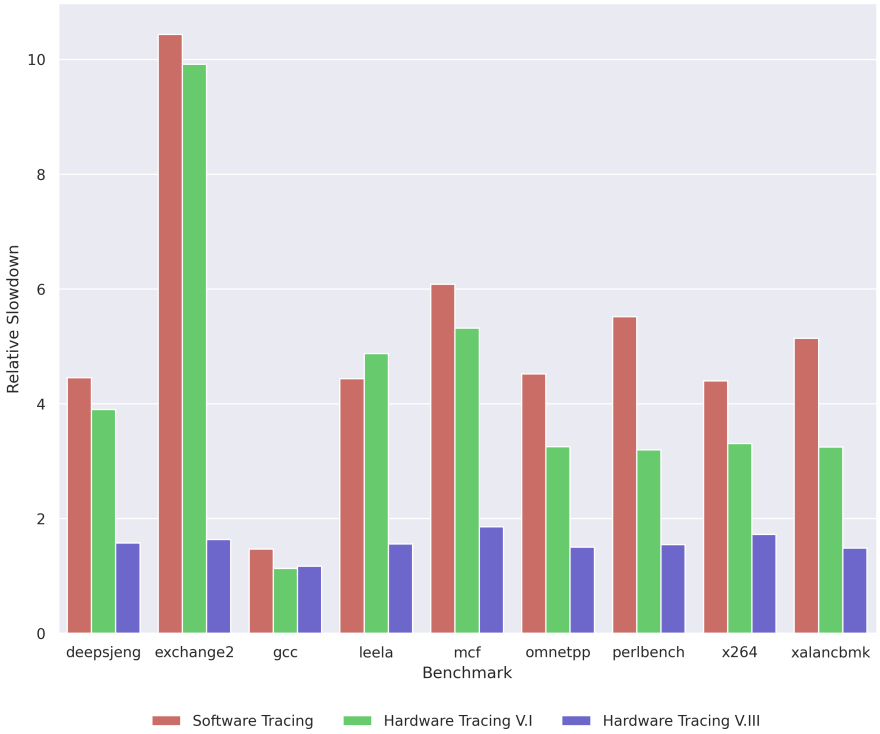


Figure 28: Relative slowdown on benchmarks from 3rd hardware tracing method

Looking at the results in Figure 28 above, it is clear that this hardware approach is significantly better than all previous software and hardware tracing methods. This does not mean that hardware is better than software. There is still the problem of parsing the produced Intel PT data into a complete trace. If the parser can be made fast enough this approach would be the best out of all previous tracing methods.

3.7 Parser

As discussed throughout the previous sections, to turn the Intel PT data into a trace, a parser is needed. This parser needs to take in the various data files generated by the hardware tracing implementations and use that to generate a trace.

To keep the parser simple, it is implemented as a second program. In future, it would be preferable for this to be implemented inside the QEMU code base. As the parser plays an integral role in the performance of the hardware tracing methods, C++ as a language was chosen as it has similar performance to c but contains higher-level data structures such as maps.

The parser is split up into three key sections. There is the mapping parser (mapping-parse.cpp); this parser reads the mapping information generated to create a map from target address to guest addresses.

Next is the asm parser (asm-parse.cpp); this parser reads the assembly code information generated and provides methods for searching and following this assembly code. Lastly is the Intel PT parser (pt-parse.cpp). This parser parses the Intel PT packet stream and uses that stream alongside the other parsers to generate a complete trace.

When designing and implementing the parser, two core ideas were paramount. Firstly, performance over simplicity, the parser needs to be as preferment as possible, even if that means sacrificing simple design. Secondly, no global state. In future, to gain more performance, concurrency may be wanted. To make that transition easier, no global state should be used for the parser. As such, functions in the parser should be pure functions as much as possible.

3.7.1 Mapping Parser

The mapping parser is the simplest of the three parsers. This parser provides two global functions: `get_mapping(u64 target_ip)` and `load_mapping_file(const char *file_name)`. All this parser needs to do is read the entire mapping file generated by QEMU and use the data of that file to implement the `get_mapping(u64 target_ip)` function.

This function takes in a target address and returns its corresponding guest address if one exists. To do this, the load mapping function reads in the mapping and uses that information to populate an internal hash map. This hash map uses target addresses as a key and guest address as a value.

Normally the C++ unordered_map [11] data structure would be used for a situation like this. However, it is not fast enough. unordered_map has strict rules around maintaining reference validity which leads to buckets filling up. As the buckets are implemented using a linked list, this can lead to poor lookup performance.

Instead of using C++'s unordered map, robin hood's unordered flat map [26] is used instead. This map offers much better performance for this scenario.

3.7.2 Assembly Code Parser

The asm parser is significantly more complex than the previous one. It provides the following global functions:

1. `void asm_init(asm_state& state, const char* asm_file_name)`
2. `void advance_to_ipt_start(asm_state& state)`
3. `bool ip_inside_block(asm_state& state, u64 ip)`
4. `instruction* get_next_instr(asm_state& state, u64 ip)`

One thing which stands out about all of these instructions is that they all take in a reference to an asm_state struct. This state struct allows the elimination of global state by passing in a reference to this state. Passing state in this way, compared to using global state, makes it much simpler to debug the interaction between functions and try out new methods. It also helps keep the functions pure, one of the requirements set out for the parsers.

The asm_init function initialises this state struct and loads in the file containing the assembly code data generated by QEMU. Recall that this assembly code data changes throughout runtime, so the parser cannot read it all in at once. Instead, the parser must wait for the advance to ipt start function to be called. When this function is called, the parser will read all assembly code information up to the next ipt start marker, and deal with it accordingly.

The third function is used to check if a given address lies inside a translated block. The Intel PT parser uses this to check if it should follow the assembly code. It can only follow assembly if it lies within a translated block. It's also useful to check for bugs. If this function returns false, but the given address has a mapping, that means the assembly code information has gotten out of sync with the programs execution.

Lastly, there is the get next instruction function. This function returns the first instruction after a given address. If the given address does not lie inside a translated block, or there is no instruction after this address, null is returned.

The initial implementation of these functions used two ordered maps. The

first is a translated block map, which maps from the target address to the size of a translated block starting at that address. The second map is an instruction map, which would map from target address to the instruction at that address.

The problem with this approach is that it is very slow. By using an ordered map, the lookup is log n complexity. Also, since all instructions were stored in a single map, every time the parser needs to get the next instruction, a log n search must be performed through every single instruction.

To improve performance, the design was iteratively improved, leading to the design shown below in Figure 29.

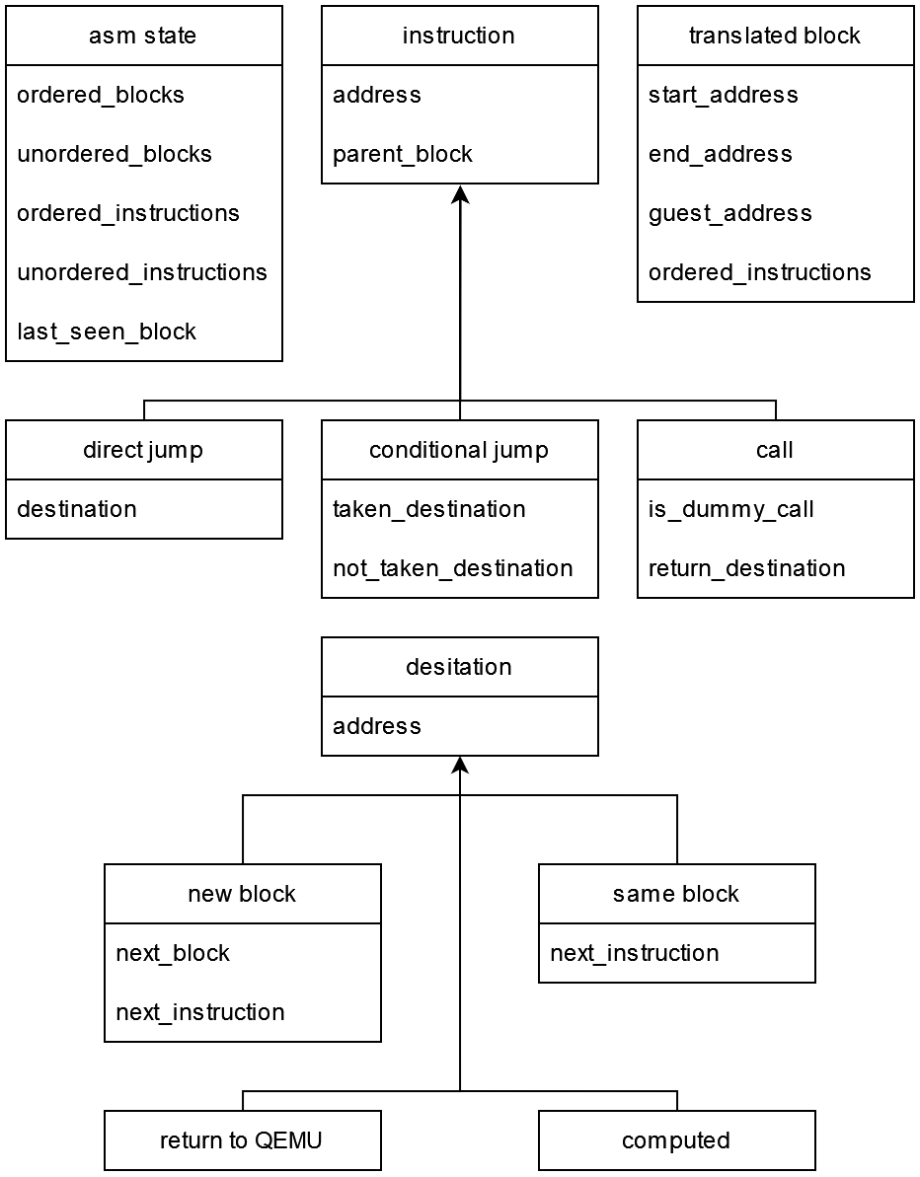


Figure 29: Assembly Code Representation

There is a lot to unpack in the above Figure 29. To start, is the new translated block design. Before, only a mapping from a translated block start address to its size was stored. Now the translated blocks are represented by a struct. Alongside the original ordered mapping there is now an unordered mapping from start address to translated block.

The translated block struct stores the start and end addresses of the translated block. It also stores the guest address for this block. Storing the guest address means that a lookup to the mapping only has to occur once for each translated block. Lastly, the block stores an unordered mapping of instructions that are contained in this block.

When the parser starts parings a new translated block, it creates a new struct and begins adding each instruction to its instruction map. When the parser finishes parsing this block, it can store the block in the ordered and unordered block maps of the asm state. It also stores a reference to all of the blocks' instructions in the ordered and unordered instruction maps within the asm state.

Storing the instructions and blocks with this method leads to a few benefits. Firstly, if given an address, to find the next instruction after that address, a lookup can be made to get the block containing that address. Then a lookup can be made to find the first instruction within that block after that address.

Both of these operations in this operation are log n which is the same as before. However, the data sets being searched are significantly smaller. Secondly, storing the start and end addresses of a block makes it easier to check if a given address belongs in that block.

The next data structure to unpack is how instructions and their destinations are represented. There are three possible instruction types which can be seen. A regular direct jump, a conditional jump, or a call instruction.

For a direct jump, the parser stores the destination of this jump. For a conditional jump, the parser stores both the taken and not taken destination. The not taken destination is simply the address after the conditional jump. This allows the Intel PT parser to use its TNT packet information to choose which destination to take. Lastly, there is a call instruction.

All the Intel PT parser needs to know about a call instruction is if it is a dummy call. Recall this is how the parser can decide to ignore a call and wait for the dummy call, or it should know that this call immediately returns to the translated block. However, by storing the return destination (next address after the call), the parser does not need to calculate the address after the call; it already has access to that information.

Before the redesign, a destination was represented only as an address (i.e. a u64). After following a direct jump, for example, the Intel PT parser would need to perform a lookup to get the next instruction after that destination address. This can lead to a lot of duplication of work. A given jump can be executed multiple times. For each execution, the Intel PT parser would need to perform the same call to look up the same next instruction. To remove this duplication of work, the destination struct was created.

There are four types of destination. The first is a same block destination. In this, the next instruction is within the same block, and a reference to that instruction is stored within this destination struct. This allows the parser to get the next instruction immediately, and since it is within the same block, the parser knows executing this jump will not add a new block to the trace, so it does not need to do any more work. Note if a jump instruction is to the start of the current block, this is not represented as the same block destination because that will add a new address to the trace.

Secondly, there is a new block destination. In this destination, the next instruction is at the start of a translated block. As such, a reference to the translated block and the first instruction within that block is stored. The Intel PT parser can use the next block reference to get the guest address and add it to the trace. It can also use the next instruction reference to get access to the next instruction immediately.

Third, there is a computed destination. In this scenario, the address of the jump is not known. This means that the assembly code data cannot provide any useful information. Instead, the Intel PT parser needs to wait for the next TIP packet to determine the result of this jump.

Lastly, there is a return to QEMU destination. If a jump within a translated block is made to code which is not within another translated block, that can only mean that the translated block is exiting out of translated code and returning back to QEMU. Nothing can be done here as the Intel PT parser needs to wait for data, signalling that execution has returned to translated code.

This redesign can lead to noticeable performance increases. For instance, before, when the destination of instruction was to the same block or another block, the Intel PT parser would need to request to get the next instruction, which would perform a log n search on all instructions. Now, the parser has instant access to that next instruction via the attributes on the destination struct.

The asm parser can also create the destination struct quicker than the previous search. If the destination is within the same block, only that block's instructions need to be searched. If it is to a new block, it must point to the start of that block. Hence the unordered map can be used, which has constant time lookup.

3.7.3 Intel PT Parser

Finlay, there is the Intel PT parser. This parser deals with reading the Intel PT data and using that in combination with the other two parsers to generate a complete trace. As this parser is the one which communicates with the others, it only has one public function, `start()`.

This function takes in the file names of all the data files produced by QEMU so it can pass them to the other parsers and then start parsing. However, it has several key functions which are essential to understanding its implementation, and those are:

1. `void parse(pt_state& state)`
2. `void handle_tip(pt_state& state)`
3. `void follow_asm(pt_state& state)`
4. `optional<pt_packet> try_get_next_packet(pt_state& state)`
5. `void update_current_ip(pt_state& state, u64 ip)`

As with the asm parser, a common thing to note about all of these functions is that they take in reference to a pt_state struct. This state struct serves the same purpose as the asm state struct served for the asm parser.

The pt state struct is significantly larger as there are a lot of edge cases which need to be tracked when dealing with Intel PT data. Two key attributes in the pt state struct are current_ip and last_packet. These attributes store the current instruction pointer value and the last Intel PT packet that was parsed.

The main function from the list above is the `parse()` function. This function loops until it has dealt with all the Intel PT data, and thus a complete trace has been generated. This function can be seen below in Figure 30.

Looking at the implementation, it shows that the function works by first reading the next packet in the Intel PT data file. This packet is stored in the state struct. The next key things are to check if this packet is a TIP packet. If it is, the address it contains needs to be extracted and handled. This is done via the `handle_tip()` function.

The handling of TIP packets is essentially a series of if statements to check what condition the parser is in and what this TIP packet means. For instance, if when following assembly, a call was executed. The parser needs to check if this TIP packet points to the address of the dummy helper call. If this is the case, then the parser knows it can start to follow the assembly code again; if not, the parser knows it must wait.

Another check is to see if this is the first TIP packet ever seen. If this is the case, it means the address within this packet is the address of the function used to start Intel PT recording; this value is important for later.

```
// pt-parse.c
void parse(pt_state& state)
{
    std::optional<pt_packet> maybe_packet;

    while((maybe_packet = try_get_next_packet(state))) {
        pt_packet packet = *maybe_packet;

        if(packet.type == PAD || packet.type == UNKOWN)
            continue;

        state.last_packet = &packet;

        // Handle this packet
        if(packet.type == PSB) {
            state.in_psb = true;
        } else if(packet.type == PSBEND) {
            state.in_psb = false;
        } else if(packet.type == TIP) {
            handle_tip(state);
        }

        // Follow asm if possible
        if(can_follow_asm(state)) {
            follow_asm(state);
        }

        // Track if the prev packet was mode / ovf
        state.last_was_mode = false;
        state.last_was_ovf = false;

        if(packet.type == MODE) {
            state.last_was_mode = true;
        } else if(packet.type == OVF) {
            state.last_was_ovf = true;
        }
    }
}
```

Figure 30: Intel PT Parse function implementation

The results of the `handle_tip()` function lets the `can_follow_asm()` function know if it should allow the assembly code to be followed. If the hardware tracing method being parsed does not generate any assembly code (the first method created), then this function will always return false to prevent any errors.

If the `follow_asm()` function is called, it will use the current_ip attribute to get the next instruction in the assembly code. It will then follow the assembly code using the methods described in the asm parsing section. If this function encounters a conditional jump, it checks to see if the last packet parsed was a TNT packet. If it was, then it can use the data within this packet to determine if the conditional jump was taken or not. If the last packet was not of type TNT, then the following assembly must stop.

One important item to note is that the current_ip attribute is never modified directly. Instead, a call to the `update_current_ip()` function is made. Whenever the current address is changed, there are a few conditions that need to be checked. Firstly, if this new address is the address of the function which is used to start Intel PT recording, then this means that the assembly code representation should be updated to the next recording start marker within the assembly code file.

Another important check which must be done is to see if the given address maps to a guest block. If it does, then the address of this guest block is saved to the output trace file. It also means assembly code can be followed in the next run of the loop.

Recall in the asm parsing section, it was discussed that performing all of these checks can have a negative impact on performance. To prevent this when updating the current_ip from assembly code there is another function `update_current_ip_from_destination()` which is used. This function can remove some of the checks and help improve performance.

Finally, to try and increase performance as much as possible, when parsing the Intel PT data into packets, the data file is read into memory in chunks (100MB in size). The chunk is parsed into packets until it runs out, after which the next chunk is read in until no data remains. This limits the number of IO operations performed.

3.7.4 Summary

In short, the parser works by combining three sub-parsers. A mapping parser reads the mapping information to create a method of getting the guest address of a given target address. The asm parser reads the assembly code information and provides methods for following that assembly code. Lastly, there is the Intel PT parser which reads the Intel PT data and uses that data alongside the other two parsers to produce a complete trace of a given program.

With the parser complete, the benchmarks for the previous hardware tracers can be re-evaluated, taking into account the time taken to parse the data generated to produce a complete trace. For the hardware tracers to be useful, their total time should be lower than the time taken to generate a trace using software methods. The results of the benchmarks can be seen below in Figure 31.

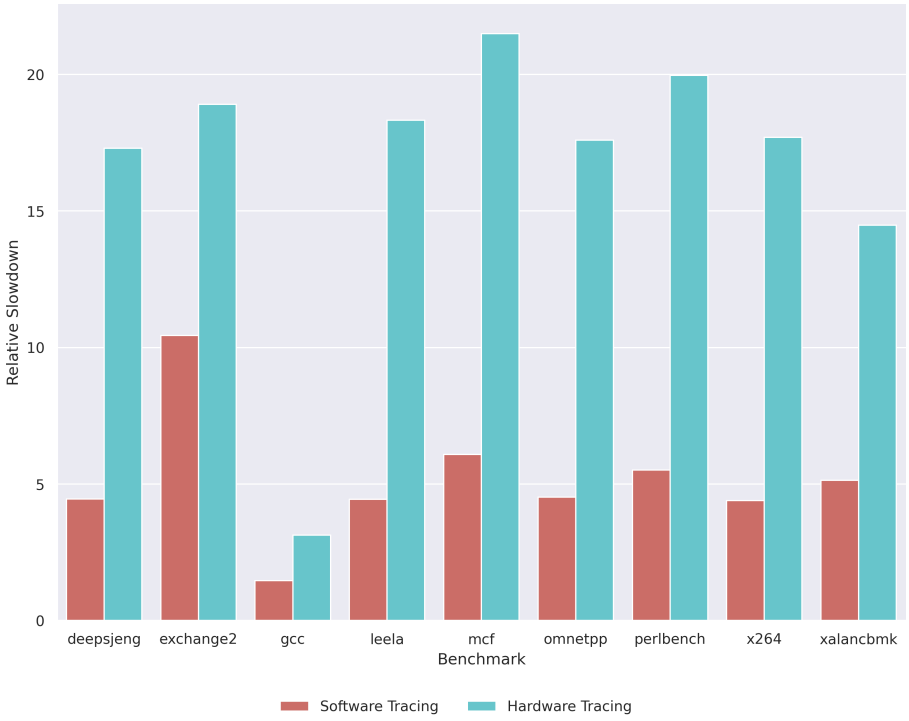


Figure 31: Relative slowdown on benchmarks from 3rd hardware tracing method with parsing time included

Looking at Figure 31 above, it is instantly apparent that the parsing is adding a huge overhead, making the once faster hardware tracer significantly slower than the software tracer. It is expected that parsing would add overhead as Intel PT generates vast amounts of data, and parsing will take time. Due to this, it appears that the hardware tracing method is not yet good enough to beat the software tracer.

The parsing time needs to be reduced, but the parser is already extremely efficient, and there isn't much which can be done to improve its runtime. One possibility would be to create another tracing method that does not rely on assembly code information. This would mean the parser does not need to deal with it, possibly improving runtime.

Before attempting to see if it is possible to implement a tracer like this, it would be useful to know if parsing and following the assembly code is slow. To do this, the benchmarks are run again, but this time the `can_follow_asm()` function is set to always return false. This will not produce a valid trace, but it will produce information about how much time is being spent following assembly code when parsing. The results of these benchmarks are shown below in Figure 31.

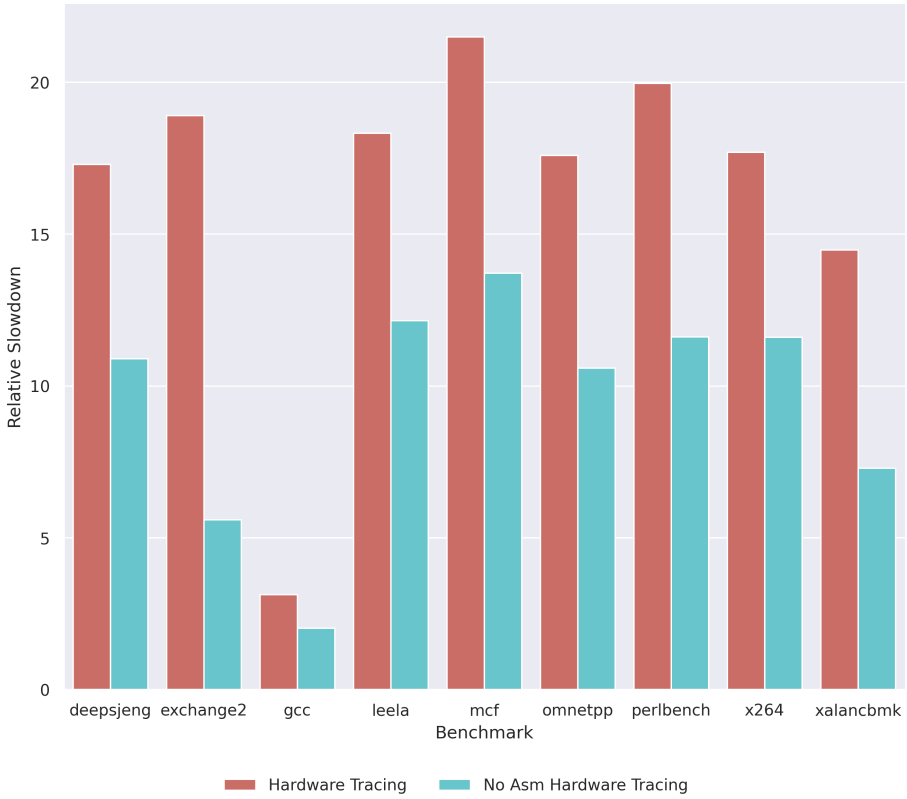


Figure 32: Relative slowdown on benchmarks from 3rd hardware tracing method with parsing time included and asm disabled

Looking at the results in Figure 32, it does show that removing the tracing of assembly code does improve parsing performance. It does not appear to be enough to beat software, but it still suggests that this is an area worth exploring.

3.8 Hardware Version IV

Following from the idea that parsing of the assembly code is slow, the goal for this tracing method is for it to be able to generate a complete trace with no assembly code required whilst also keeping direct chaining fully enabled. If both of these requirements are met, this method should be the most performant.

To achieve this goal, QEMU must force Intel PT to generate a TIP packet when a translated block is executed. This would mean the parser does not need any assembly code information and could generate a complete trace. Another possible solution could be to use the PTWRITE functionality of Intel PT. This allows data to be written directly to the Intel PT buffers. However, this feature was not present on the hardware used for development; thus, it cannot be explored.

Recall that Intel PT generates a TIP packet whenever a computed jump is executed. For example, when a QEMU helper function is called, this is implemented as a computed jump, causing a TIP packet to generate. If every translated block could be given its own helper function to call when executed, it would provide the parser with a unique TIP packet for each translated block allowing it to generate a trace.

Creating this many helper functions is not feasible, however, something similar can be done. A call instruction can be inserted at the start of every translated block. This call will call the instruction preceding itself. This call can be turned into a computed call by placing the address of the next instruction into a register and calling the value of that register.

This process will not change the functionality of the translated block, but it will force a TIP packet to get produced. The exact assembly code inserted at the start of every translated block is shown below in Figure 33. Note the `addq` instruction is required to offset the effect a call instruction has.

```
leaq  2(%rip), %rax
callq  *%rax
addq  $8, %rsp
```

Figure 33: Computed Call Assembly

To add this assembly code to each basic block, the `tcg_gen_code()` function is modified to insert this assembly code into the translation block before the translation of the guest block begins. All the assembly code information generated in the previous hardware method can be ignored.

The mapping information (mapping from target to guest address) is still needed. However, the mapping will be off. The address generated in the TIP packets produced by the inserted call instructions will be nine higher than the recorded mapping. The reason for this offset is that the combined size of the `leaq` and `callq` instructions are nine bytes. To account for this difference, a constant value of 9 is added to every target address in the mapping.

As with the previous tracing versions the benchmarks can be run again to evaluate the performance of this method. The result of these benchmarks are shown below in Figure 34.

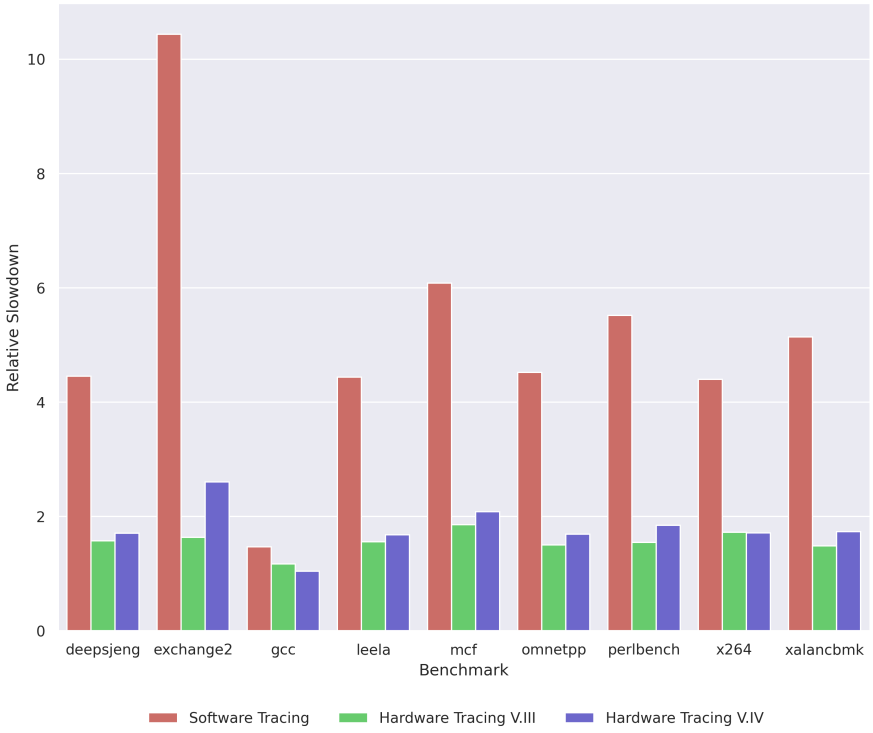


Figure 34: Relative slowdown on benchmarks from 4th hardware tracing method

Looking at the results of the benchmarks in Figure 34, it is clear that this new method is still significantly faster than the software approach. It does appear to be slightly slower than the previous hardware approach, but if the removal of the need to parse assembly code improves parsing time, this change will be worth it.

To determine the effect on parsing time the benchmarks were run again with parsing included. The results of this can be seen below in Figure 35.

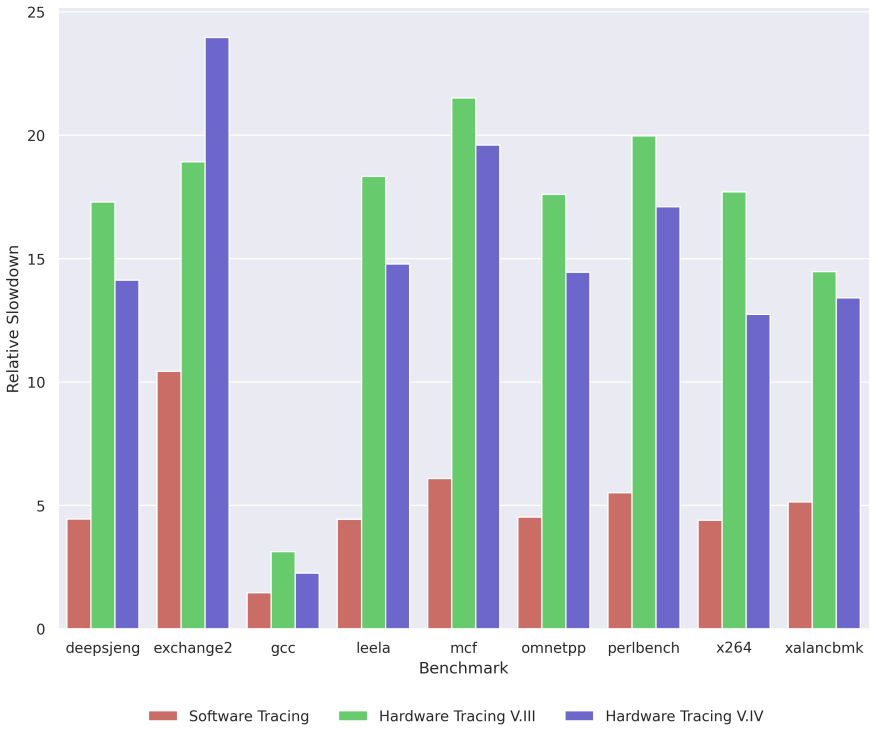


Figure 35: Relative slowdown on benchmarks from 4th hardware tracing method with parsing time

When looking at the results in Figure 35, it shows that this final Hardware method is faster than the previous in most benchmarks. With the exception being exhange2, for which the previous hardware method is faster. Though this method is faster, it is only slightly faster. This hardware method is also not faster than the best software method. The software method still produces the best results on all benchmarks by a significant margin.

4 Evaluation

To evaluate the various tracing implementations benchmarks from the Standard Performance Evaluation Corporation [29] (SPEC) was used. specifically the following benchmarks from the SPEC CPU® 2017 benchmark package.

- 531.deepsjeng_r (deepsjeng) [7]
- 548.exchange2_r (exchange2) [9]
- 502.gcc_r (gcc) [2]
- 541.leela_r (leela) [8]
- 505.mcf_r (mcf) [3]
- 520.omnetpp_r (omnetpp) [4]
- 500.perlbench_r (perlbench) [1]
- 525.x264_r (x264) [6]
- 523.xalancbmk_r [5]

During the design and implementation phase, these benchmarks were run twice, and the results were averaged. For the final evaluation, the results were run a total of ten times, with the results being averaged. For each benchmark, the time taken for QEMU to finish executing was recorded. If the benchmark required parsing, the time for parsing was recorded. Alongside parsing time, the amount of storage space required for data being supplied to the parser was also recorded.

In the final evaluation phase, only the second software and the final two hardware implementations are evaluated.

4.1 Runtime Performance

As with the evaluation done throughout the design and implementation section, the first area to look at is QEMU runtime (parsing not included). The results of the benchmarks can be seen below in Figure 36.

This figure shows the slowdown each tracing method had on QEMU runtime. Meaning for each benchmark, the value of 'Without Tracing' is always one. For the other tracing methods, the value is the multiple of the original it took to complete. For example, looking at Figure 36, for the mcf benchmark, the software tracing method has a value of six. This means it took six times longer to complete compared to running with no trace being generated.

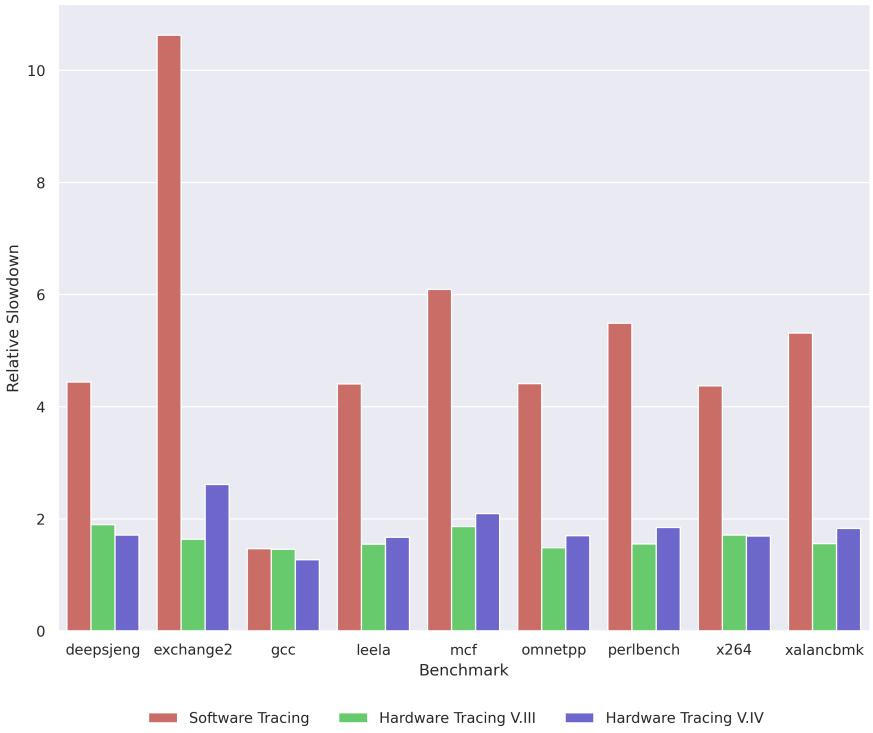


Figure 36: Relative slowdown on benchmarks

The results in Figure 36 reinforce the analysis performed during the initial design phase. When looking at QEMU runtime, the hardware versions massively outperform software tracing methods. Again as expected, hardware version four is slightly slower than version three. This slowdown is due to the extra call instructions added to each translated block.

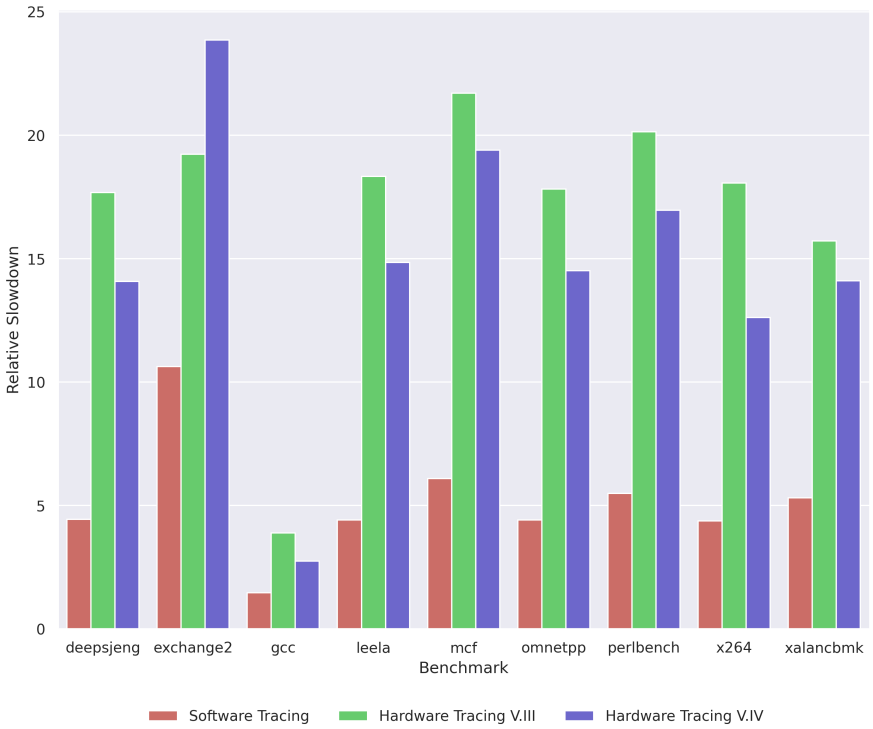


Figure 37: Relative slowdown on benchmarks with parsing time

Figure 37, above shows the benchmark results with parsing time included. Again these results are expected and match what was discovered during the design phase. When parsing is included, the hardware versions are significantly worse than software tracing.

4.2 Storage Performance

One area worth evaluating which hasn't been discussed up to this point is the storage use of the various implementations. The software tracing methods only produce the final trace. However, the hardware methods produce intermediate data, which is parsed into a final trace. This intermediate data can take up a significant amount of storage space.

To evaluate the storage use after each benchmark was run, the amount of space required to store the intermediate data was logged. This information was used to produce Figure 38, below. This figure shows the amount of data generated per second of runtime for each benchmark, where the runtime is the time taken for the benchmark to execute with no trace being generated.

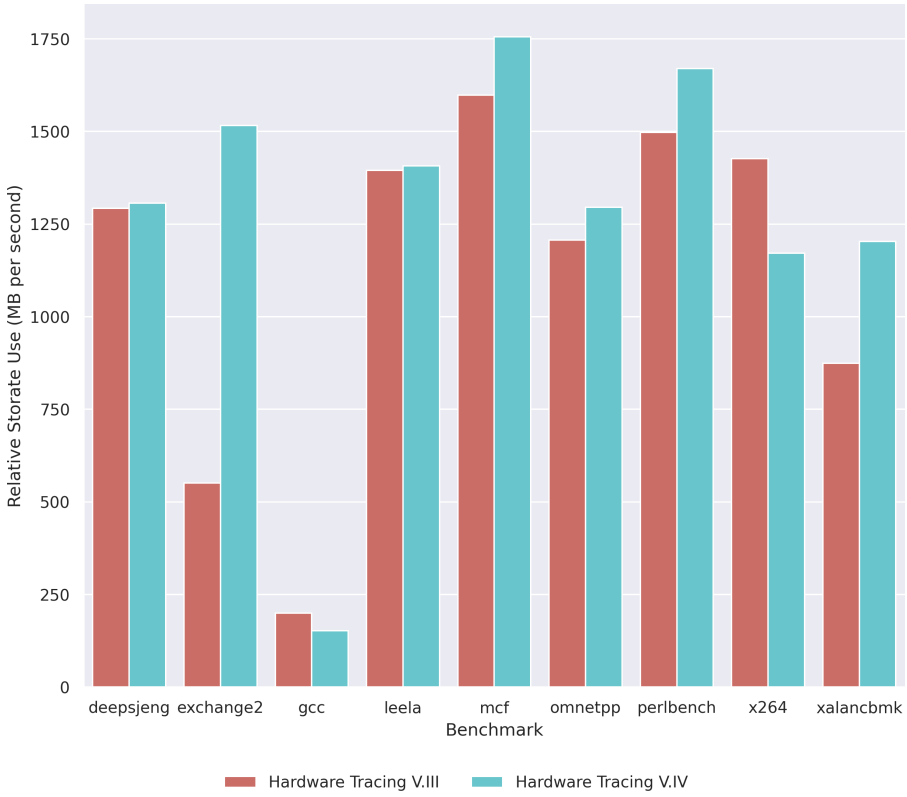


Figure 38: Relative storage per benchmark

Looking at the results in Figure 38, it shows that for most of the benchmarks, both hardware versions produce between 1 and 1.5GB of data per second, with hardware version 4 using slightly more data.

It is expected that version 4 would generate more data. Though it does not need the assembly code data, the additional TIP packets being produced will increase the amount of Intel PT data being generated by a noticeable amount.

One thing which isn't clear from the graph is how much data is being produced. If the program being run takes minutes to complete, the amount of data being produced is in the realm of 100s of GB. This makes tracing long-running programs with hardware not feasible, without significant storage space.

As with runtime, the outlying benchmarks exhcange2 and gcc are also outliers in terms of storage use. This is expected and also helps explain the outlying parsing times associated with these benchmarks.

Looking at the results for the exchange2 benchmark. Hardware version 4 produces significantly more storage than version 3 when running the same benchmark. Since the exchange2 utilises more patch chaining, this is expected.

Recall the patch-chaining method uses direct jumps to chain together two translated blocks. As such, no Intel PT data is needed to inform the parser of this. However, hardware version 4, forces Intel PT to create a TIP packet in this situation.

At a minimum, a single TIP packet is two bytes in size. Though this isn't large, two bytes is much more than none. As such, with programs that contain a large amount of patch chaining hardware, version 4 will generate a lot more data than version 3.

The other outlier is gcc. Recall that the gcc benchmark is a short-running program. Most of the runtime is spent starting up and cleaning up the QEMU runtime. Hence such a small amount of data is generated per second of runtime.

4.3 Usability

One final area to evaluate is the usability of all the tracing methods. If a user wants to generate a trace, they must use the '-pt-trace' argument added to QEMU. This argument takes a number as input signifying which tracing method to use. 0 represents software version I, 1 software version II, and so on, with 5 representing hardware version IV. For example, to generate a trace using software version I, the user would run the following command:

```
$ ./qemu-aarch64 -pt-trace 0 ~/programs/gcc
```

For the software versions, a complete trace is generated and stored in ∼/pt-trace-data. The user can change this location by using the '-pt-loc' argument. This argument takes the location of a folder to store the resulting trace in. For the hardware versions, the output isn't a complete trace. Instead, it outputs the series of files needed by the parser to generate the complete trace. Alongside those files, it generates an info file telling the compiler what pt trace version was used.

If no arguments are supplied to the parser, it will attempt to load the data from the default trace location. If the data is not there, then it needs to be told the correct location. The parser will store the resulting trace in the same folder. So, for example, if a user wanted to generate the same trace as seen above but using a hardware tracing method, then the following commands would need to be run.

```
$ ./qemu-aarch64 -pt-trace 2 ~/programs/gcc
$ ./parser
```

Or if using a non-default location, then the following commands.

```
$ ./qemu-aarch64 -pt-trace 2 -pt-loc ~/path ~/programs/gcc
$ ./parser ~/path
```

Though both hardware and software are easy to use, it is definitely easier to use the software approach. There is no need to worry about the second parsing program. In future, the goal would be to have the parser brought into QEMU, but currently, it is defiantly harder to use.

4.4 Future Work

4.4.1 Minor Parser Improvements

The most pressing area of future work is to improve the parsing time. The parser itself has been quite heavily optimised and will need a serious redesign to get significant performance improvements. However, there are still smaller incremental changes which could lead to performance gains.

One area worth investigating is the idea of a hot and a cold target to guest address map. Recall that the parser contains a single map, mapping from target address to guest address. For large programs, this map will contain a significant number of entries.

Instead of having a single map populated with all entries, there could be a hot map that contains the most recently used entries and a cold map containing all other entries.

When searching for a mapping, first search the hot map, then if it cannot be found, search the cold map. If an entry is found, it is marked as hot and moved into the hot map. There can be a limit on the number of elements in the hot map, and once that has been reached, an algorithm (least recently used, for example) can be used to remove elements back into the cold map.

This method, in theory, would benefit both hardware versions three and four; however, it is likely to improve version 4 mostly. Recall that version 3 relies on the internal assembly code representation. The structs used in this representation contain the guest address already, meaning fewer lookups are made.

Other small design changes were investigated but did not immediately lead to performance improvements. In future, these could be explored further. One idea was to compress TNT packets, though this improvement will only benefit hardware version three as four does not use TNT packets..

When parsing, if the parsers see a contiguous list of TNT packets, they are combined into one larger packet. The goal of this was to allow the following assembly loop in the parser to continue for longer before having to exit and parse more packets. The thought behind this is that doing more parsing, followed by more assembly code following, instead of switching between the two, would lead to greater cache utilisation.

Another area which could be investigated is modifying how the assembly code information is outputted. Currently, this information is not outputted into the file in order. Meaning that for a given block, the instructions outputted for that block are in a different order in which they appear in memory. It is possible to fix this issue. This would make the parsing of assembly code information easier possible leading to performance gains.

The way in which the parser stores the assembly code instructions could also be improved. Currently, all instructions are heap-allocated individually and stored in their respective translated blocks via reference. This could be changed so that the translated blocks contain an array of instructions.

An array could lead to performance improvements by keeping translated blocks and their instructions in the same region of memory, leading to better cache utilisation. The difficulty of this approach is that the translated blocks would become a non-standard-sized struct. Both the assembly code improvements would only benefit hardware version three.

4.4.2 Parallel Parser

The biggest potential for performance improvements in the parser is to make it parse the data in parallel. Before this can be explored a brief introduction of parallel patterns is warranted.

First, there is some terminology that must be explained. A parallel task is a unit of work, or computation, to be performed in parallel [18]. These tasks can be supplied as jobs to a parallel farm (or tasks queue). A parallel farm has a series of workers which operate in their own threads. These workers take a job of an input queue, process it and put the result into an output queue.

The parsing for the third and fourth hardware implementations could be made to work in parallel, but the fourth has the most potential for speedup. The difficulty with the third implementation is that it requires assembly code information alongside the Intel PT data. Thus, the parsing of this information would need to work in parallel.

Recall, that the assembly code changes throughout the runtime. This means the parser cannot jump to a given location in the assembly code file and start parsing; it must parse it from the start. Having to parse the assembly code from the start means it cannot be done in parallel, making the third hardware method not suitable for parallel parsing.

The fourth hardware parsing method does not use any assembly code information. This method relies only on the Intel PT data, and the mapping information (which is constant, so it can be shared amongst all worker threads). TIP packets do rely on the past Intel PT data to be generated.

Intel PT compresses TIP packets, by utilising the previous TIP packet value. If, for instance, the first byte of the address in a TIP packet is the same as the first byte in the previous TIP address, then this byte won't be included in the packet. This appears to be the same problem as with parsing assembly code; however, the solution is the use of Packet Stream Boundary (PSB) packets.

PSB packets serve many purposes but essentially act as a reset point for a parser. When the parser sees a PSB packet, it waits for a PSBEND packet. In between these two packets will be a full non-compressed TIP packet. This allows the parser to get the exact location of execution, even if data was lost prior to the PSB packet. The parser can also reset its return stack and other information, but this isn't needed for tracing purposes.

From this, the Intel PT data can be thought of as a series of chunks separated by PSB packets. Each of these chunks is indenedent meaning they can be parsed in isolation to produce part of the complete trace. The complete trace can be obtained by combining all of these parts.

A possible solution to implement this concurrent parser is to find the location of all the PSB packets. Then the file can be split into chunks which can get supplied as jobs to farm. However, this solution defeats the purposes of the concurrency; the parser has to linearly parse the entire file first to get the PSB locations, before starting the concurrent work.

A better solution is to split the Intel PT data into N equal-sized parts. These parts can be supplied as jobs to the concurrent farm. When the worker receives a job, it looks for the first PSB packet in that job. After this, the worker can start parsing, producing a section of the complete trace. The worker will keep parsing until it finds the first PSB packet after the end of its given job.

The worker keeps parsing past the end of its job because it needs to parse up to the first PSB packet in another worker's job. Recall the workers ignore all data before the first PSB packet in their job.

In practice, this method means each worker is given a start and end location in the Intel PT data file. The worker will produce a valid trace for that section of the Intel PT data file. After the entire file has been parsed, these sections can be combined together to produce a complete trace.

In theory, this method could work very well. All of the work is being performed in parallel, the parser has complete control of the size of the jobs given to its workers, allowing for an optimal size to be chosen to minimise the proportion of time spent adding jobs to queues and starting up threads.

If this parser was implemented, take the mcf benchmark. It took, on average, 867 seconds to parse the Intel PT data. Assuming this was split into six even jobs and parsed in parallel on a six-core machine, it could take an average of 145 seconds for each worker to complete.

Even if there were a 20% overhead due to workers taking uneven time to complete and the time needed to start threads, it would reduce parsing time down to 173 seconds. Which, when combined with the time taken for the hardware version 4 to finish execution, would make it faster than the software version tracing method.

These numbers are very rough, and it would require a more detailed investigation. Still, it does appear that implementing a parallel parser could make the hardware version faster than the software version.

4.4.3 Moving Parser into QEMU

To improve usability and increase performance, the parser needs to be moved into QEMU. Prior to doing this, the parallel parser should be completed, and it is that parser which can be brought into QEMU.

To bring the parser into QEMU, another thread of execution needs to be created (alongside the tracing thread that records the Intel PT Data). This thread waits for the tracing thread to save data. As it is saved, this thread will split it into chunks and pass those chunks as jobs to the parallel parser described in the above section.

This method will only work if the machine running has enough cores, four minimum. One core for QEMU, another for the tracing thread, one for splitting the data into chunks, and a final for the parsing farm.

This method will increase performance, as currently, the parser needs to wait for all data to be generated before it can begin. This is not necessary, leading to wasted time. This method could also be used to minimise the amount of storage space needed. After data has been parsed, it is not needed anymore and can be deleted.

To get as much performance with this method as possible, when the program has finished exiting, and all data has been saved out of the buffers. Two more workers can be added to the parsing farm. This is possible because the QEMU and the Intel PT recording thread have now finished, freeing up two more cores for parsing.

If the CPU which is being used has enough cores with this method, it is possible that all parsing can be finished by the time QEMU has finished executing. Apart from the final chunk of data which won't exist until QEMU executes the last block. If this is possible, then this hardware method will outperform the software version significantly.

4.4.4 Improvements Within QEMU

There are a lot of small areas for improvement within the QEMU codebase, which could reduce the overhead introduced by the hardware tracing methods.

An area which should be investigated further is the chain count. Recall that the chain count limits the number of basic blocks which can be executed at a single time via direct chaining. This is needed to prevent the Intel PT buffers from filling up, causing data loss. Currently, the chain count is set to a fixed number. This number has to be very conservative to ensure no data loss.

One possible replacement is to have the chain count be a boolean, with true indicating execution should stop and false otherwise. The Intel PT data recording thread can set the chain count to true if the recording thread sees that the buffers are nearing their limit. This will cause the next translated block to return to QEMU.

The problem with this approach is it requires the recording thread to have a continuously updating value of how much data is within the buffers, which is currently not the case. Another possible replacement is to have a dynamic chain count value.

The chain count could start small and be increased over time until a value is reached that utilises the maximum amount of the available buffer space. There is a possibility with this approach that the algorithm will overshoot and use more data than possible.

To deal with the possibility of the algorithm using more data than possible, it will need to have a method of reverting execution. The algorithm will need to know the last CPU state in which Intel PT data was successfully recorded.

If an overshoot occurs, QEMU needs to revert the chain count and the CPU state to their old values. Also, the Intel PT data generated during this process needs to be removed. This is a non-trivial design and may not even be faster as the resetting of the CPU state could take time, so it should only be investigated after all other avenues have been explored.

Another area worth investigating is how Intel PT data is collected within QEMU. An initial area to look into is setting the aux buffer to ring mode, not linear.

A ring buffer was only looked into briefly, but it did appear that a ring buffer may have a continuously updating head value. This would allow the implementation of the boolean chain count described earlier.

The problem with the ring buffer is it is harder to determine where new data begins, and old data ends. Also, during development, the ring buffer would not use more than a single page of memory when compared to the 1024 pages used in linear mode.

Currently, only the Linux perf API is used for collecting Intel PT. This is not the only method; there exists a program called simple-pt [28]. Simple-pt is a standalone program that allows for the recording of Intel PT data. It was not used as it requires kernel privilege to run, but it may solve some of the problems associated with the Linux perf API.

Lastly, the spin lock used to wait for the copying of Intel PT data could be improved. It could be replaced with a proper locking mechanism to allow for more fine-grain control. This could allow the chain count to be made less conservative without the possibility of data loss.

Recall the chain count has to be conservative, as there is a chance QEMU will miss the lock being set by the tracing thread. By using a proper locking mechanism, the possibility of missing the lock being set is removed. However, this may introduce more overhead, making it not worth it. So it is an area which needs to be looked into further.

4.4.5 Summary

To summarise, there is a lot of future work to be done. The most pressing of this future work is to make the parser utilise a parallel farm to increase parsing performance drastically. In theory, this will make hardware version 4 faster than the software tracing methods. Once this has been completed, the parser should be brought into QEMU to both increase performance and usability.

If the above two features can be completed, this should produce a fast and usable tracing method far better than what has been seen before. If further performance increases are sought, there are several more areas which can be explored. The most pressing would be to refine the method of collecting Intel PT data.

4.5 Evaluation Against Objectives

To summarise against the original objects set for this project. The first two primary objects, two implement trace generators in QEMU with and without Intel PT, were met. The second object to analyse and compare these implementations was also completed. Though given more time, a more detailed evaluation could be performed.

Specifically, given more time, a more statistically sound performance evaluation could be performed using stabilizer [12]. This paper provides a compiler system that enables statistically rigorous performance evaluations. In future, it would be worthwhile to use this to evaluate hardware implementations. It would also be useful to use this to perform a deeper evaluation of the parser to search for possible areas of speedup.

Lastly, the final second objective was partially met. The introduction of the chain count variable and the lock to prevent QEMU from executing translated code provide the functionality required by this objective. However, it is more of a static method of slowing down execution. In future, more work could be done to improve this to be more dynamic.

5 Conclusion

The key objective of this project was to implement a trace generation method into a dynamic binary translator (DBT) through the use of Intel Processor trace (Intel PT). Due to Intel PT being a relatively new feature in Intel CPUs, this was a novel approach and there isn't significant research into the possibility of using Intel PT for this purpose.

The main achievement of this project is that it showed that the use of Intel PT is possible to create a program trace through modifications to QEMU an open-source DBT. However, this Intel PT tracer was not as performant as purely software approaches. The limitation of Intel PT is the large amount of data that it generates which takes significant time to parse.

This project explored the limitations of Intel PT and produced a design that could overcome these problems. In future, through the use of a parallel parsing method, it would be possible for the Intel PT method to outperform a purely software approach. Furthermore, there are additional performance improvements which can be made to the Intel PT method in terms of data collection and parsing speeds.

Purely software tracing methods do not have many, if any, avenues to explore for performance increase. So, though the Intel PT tracing method is not faster now, in future it is very possible it will be.

References

[1] *500.perlbench_r SPEC CPU®2017 Benchmark Description*. Last accessed 29 December 2022. 2017. URL: https://www.spec.org/cpu2017/Docs/benchmarks/500.perlbench_r.html.

[2] *502.gcc_r SPEC CPU®2017 Benchmark Description*. Last accessed 29 December 2022. 2017. URL: https://www.spec.org/cpu2017/Docs/benchmarks/502.gcc_r.html.

[3] *505.mcf_r SPEC CPU®2017 Benchmark Description*. Last accessed 29 December 2022. 2017. URL: https://www.spec.org/cpu2017/Docs/benchmarks/505.mcf_r.html.

[4] *520.omnetpp_r SPEC CPU®2017 Benchmark Description*. Last accessed 29 December 2022. 2017. URL: https://www.spec.org/cpu2017/Docs/benchmarks/520.omnetpp_r.html.

[5] *523.xalancbmk_r SPEC CPU®2017 Benchmark Description*. Last accessed 29 December 2022. 2017. URL: https://www.spec.org/cpu2017/Docs/benchmarks/523.xalancbmk_r.html.

[6] *525.x264_r SPEC CPU®2017 Benchmark Description*. Last accessed 29 December 2022. 2017. URL: https://www.spec.org/cpu2017/Docs/benchmarks/525.x264_r.html.

[7] *531.deepsjeng_r SPEC CPU®2017 Benchmark Description*. Last accessed 29 December 2022. 2017. URL: https://www.spec.org/cpu2017/Docs/benchmarks/531.deepsjeng_r.html.

[8] *541.leela_r SPEC CPU®2017 Benchmark Description*. Last accessed 29 December 2022. 2017. URL: https://www.spec.org/cpu2017/Docs/benchmarks/541.leela_r.html.

[9] *548.exchange2_r SPEC CPU®2017 Benchmark Description*. Last accessed 29 December 2022. 2017. URL: https://www.spec.org/cpu2017/Docs/benchmarks/548.exchange2_r.html.

[10] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 2002. ISBN: 9780521820608.

[11] *c++ reference, std::unordered_map*. Last accessed 27 December 2022. URL: https://en.cppreference.com/w/cpp/container/unordered_map.

[12] Charlie Curtsinger and Emery D. Berger. "TABILIZER: Statistically Sound Performance Evaluation". In: (2013).

[13] John L. Hennessy David A Patterson. *Computer Architecture – A Qualitative Approach*. Katey Bircher, 2017. ISBN: 9780128119051.

[14] *HLASM Language Reference for ZOS*. Last accessed 21 December 2022. 2022. URL: https://www.ibm.com/docs/en/zos/2.1.0?topic=hlasm-language-reference.

[15] *Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 3*. Last accessed 26 December 2022. Dec. 2022. URL: https://cdrdv2-public.intel.com/671200/325462-sdm-vol-1-2abcd-3abcd.pdf.

[16] Michael Matz et al. *System V Application Binary Interface. AMD64 Architecture Processor Supplement*. Last accessed 23 December 2022. July 2012. URL: https://refspecs.linuxfoundation.org/elf/x86_64-abi-0.99.pdf.

[17] Bojan Mihajlović, Željko Žilić, and Warren J. Gross. "Dynamically Instrumenting the QEMU Emulator for Linux Process Trace Generation with the GDB Debugger". In: *ACM Transactions on Embedded Computing Systems* 13 (2014), pp. 1–18.

[18] Maurice Herlihy Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008. ISBN: 9780123705914.

[19] *Online x86 Instruction Reference: ret*. Last accessed 21 December 2022. 2022. URL: https://www.felixcloutier.com/x86/ret.html.

[20] *Perf tools support for Intel Processor Trace*. Last accessed 26 December 2022. URL: https://perf.wiki.kernel.org/index.php/Perf_tools_support_for_Intel%5C%C2%5C%AE_Processor_Trace.

[21] *perf-intel-pt Linux manual Page*. Last accessed 21 December 2022. 2022. URL: https://man7.org/linux/man-pages/man1/perf-intel-pt.1.html.

[22] *perf_event_open(2) — Linux manual page*. Last accessed 26 December 2022. URL: https://man7.org/linux/man-pages/man2/perf_event_open.2.html.

[23] *QEMU Documentation Page*. Last accessed 17 December 2022. 2022. URL: https://www.qemu.org/docs/master/about/index.html.

[24] *QEMU Supported Platforms*. Last accessed 17 December 2022. 2022. URL: https://wiki.qemu.org/Documentation/Platforms.

[25] *QEMU Translator Internals Direct Block Chaining*. Last accessed 22 December 2022. 2022. URL: https://www.qemu.org/docs/master/devel/tcg.html#direct-block-chaining.

[26] *robin-hood-hashing*. Last accessed 27 December 2022. URL: https://github.com/martinus/robin-hood-hashing.

[27] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 2015. ISBN: 9780124104099.

[28] *Simple PT*. Last accessed 3rd January 2023. URL: https://github.com/andikleen/simple-pt.

[29] *SPEC Home Page*. Last accessed 29 December 2022. URL: https://www.spec.org/.

A Ethics Self Assessment

UNIVERSITY OF ST ANDREWS
TEACHING AND RESEARCH ETHICS COMMITTEE (UTREC)
SCHOOL OF COMPUTER SCIENCE
PRELIMINARY ETHICS SELF-ASSESSMENT FORM

This Preliminary Ethics Self-Assessment Form is to be conducted by the researcher, and completed in conjunction with the Guidelines for Ethical Research Practice. All staff and students of the School of Computer Science must complete it prior to commencing research.

This Form will act as a formal record of your ethical considerations.
Tick one box

- ☐ **Staff Project**
- ☐ **Postgraduate Project**
- ☒ **Undergraduate Project**

Title of project

Intel PT for Reconstructing Guest Control Flow in a DBT

Name of researcher(s)

Robbie Wallace

Name of supervisor (for student research)

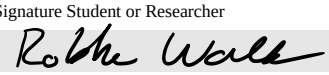
Tom Spink

OVERALL ASSESSMENT (to be signed after questions, overleaf, have been completed)

Self audit has been conducted **YES** ☒ **NO** ☐

There are no ethical issues raised by this project

Signature Student or Researcher



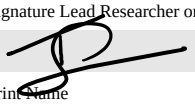
Print Name

Robbie Wallace

Date

23/09/22

Signature Lead Researcher or Supervisor



Print Name

Tom Spink

Date

23/09/22

This form must be date stamped and held in the files of the Lead Researcher or Supervisor. If fieldwork is required, a copy must also be lodged with appropriate Risk Assessment forms. The School Ethics Committee will be responsible for monitoring assessments.