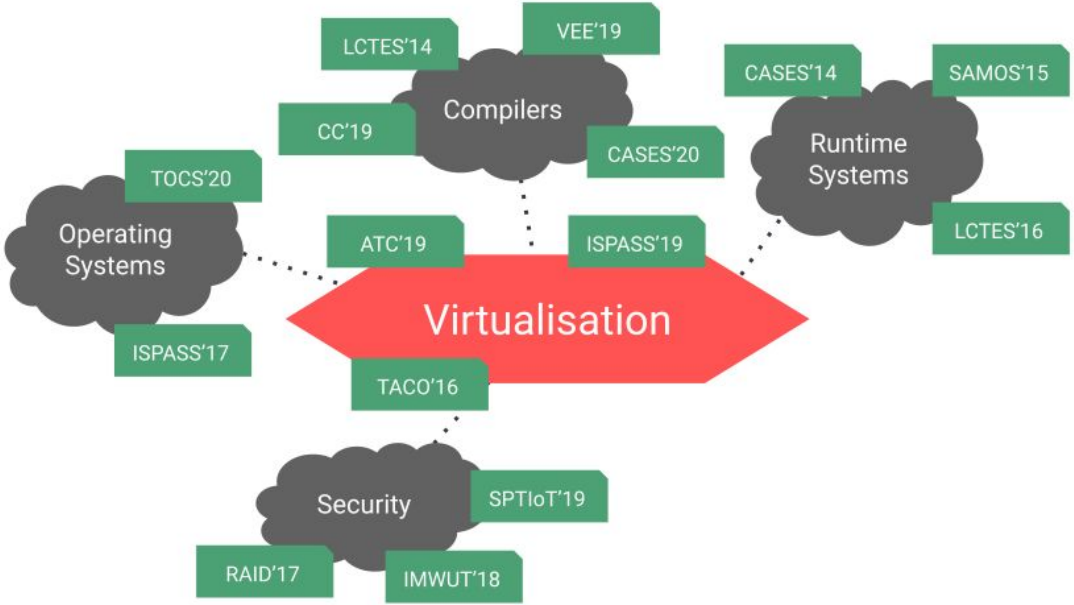
Hardware Accelerated Cross-architecture Execution Tracing

Tom Spink

University of St Andrews

About Me

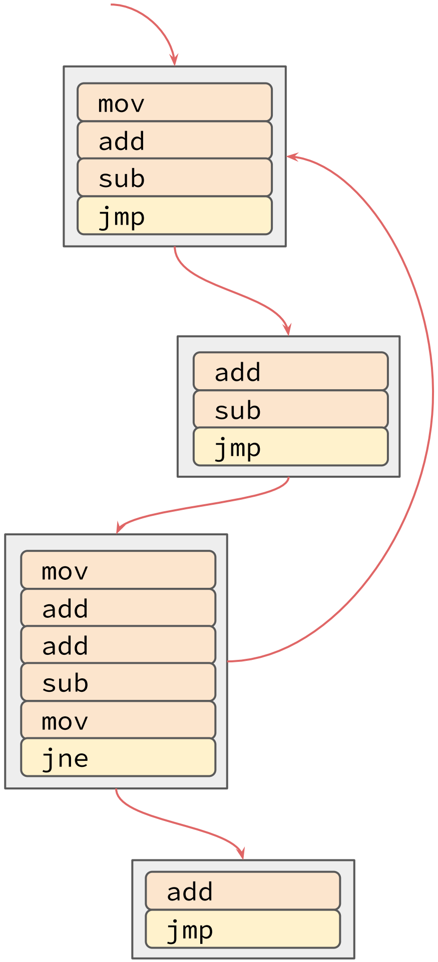
- **Lecturer** at the **School of Computer Science**, **University of St Andrews**
- Generally work in the area of **Dynamic Binary Translation**
- Core research interests:
    - **Virtualisation**
    - Operating Systems
    - Compilers
    - Runtime Systems
    - Security



Execution Tracing

What is execution tracing?

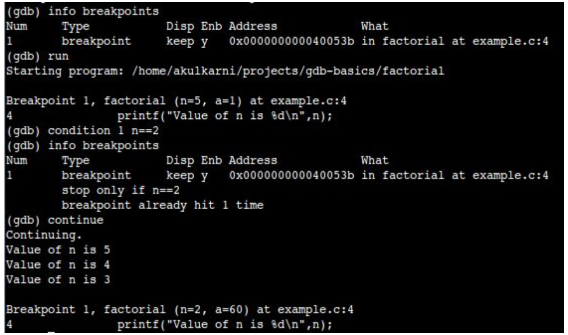
- **Running** a program, and generating a **list of instructions** that have been **executed**, during the program's **run**.
- **Granularity:** could be basic-blocks instead of instructions
- **Output:** Might generate a **control-flow graph**



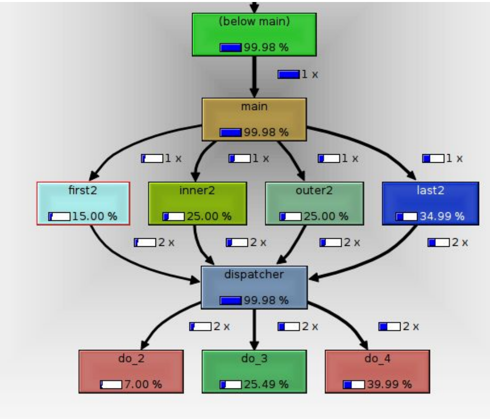
```
mov
add
sub
jmp
add
sub
jmp
mov
add
add
sub
mov
jne
mov
add
sub
jmp
add
sub
jmp
mov
add
add
sub
mov
jne
add
jmp
```

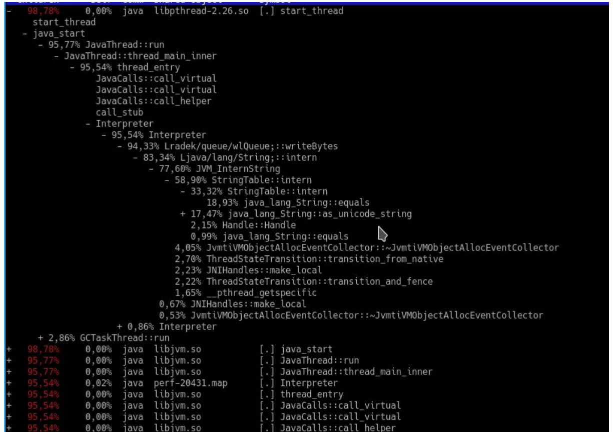
Software-based Approach

- Tools **designed** for the job:
  - `callgrind` (`valgrind`)
  - `perf`
- Tools that **can** do the job:
  - `gdb`
- Tools that give **you** the ability to do the job:
  - Intel PIN
  - Mambo-64
  - `gcc -finstrument-functions`
  - Software interrupts (`int3`)
- **Slow!**



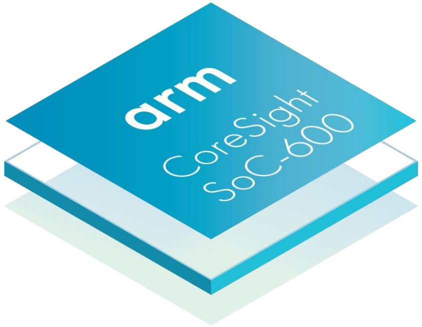
```
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000040053b in factorial at example.c:4
(gdb) run
Starting program: /home/akulkarni/projects/gdb-basics/factorial

Breakpoint 1, factorial (n=5, a=1) at example.c:4
4                 printf("Value of n is %d\n",n);
(gdb) condition 1 n==2
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000040053b in factorial at example.c:4
        stop only if n==2
        breakpoint already hit 1 time
(gdb) continue
Continuing.
Value of n is 5
Value of n is 4
Value of n is 3

Breakpoint 1, factorial (n=2, a=60) at example.c:4
4                 printf("Value of n is %d\n",n);
```





```
-   98,78%     0,00%  java  libpthread-2.26.so  [.] start_thread
     start_thread
   - java_start
      - 95,77% JavaThread::run
         - JavaThread::thread_main_inner
            - 95,54% thread_entry
                 JavaCalls::call_virtual
                 JavaCalls::call_virtual
                 JavaCalls::call_helper
                 call_stub
               - Interpreter
                  - 95,54% Interpreter
                     - 94,33% Lradek/queue/wlQueue;::writeBytes
                        - 83,34% Ljava/lang/String;::intern
                           - 77,60% JVM_InternString
                              - 58,90% StringTable::intern
                                 - 33,32% StringTable::intern
                                      18,93% java_lang_String::equals
                                 + 17,47% java_lang_String::as_unicode_string
                                   2,15% Handle::Handle
                                   0,99% java_lang_String::equals
                                4,05% JvmtiVMObjectAllocEventCollector::~JvmtiVMObjectAllocEventCollector
                                2,70% ThreadStateTransition::transition_from_native
                                2,23% JNIHandles::make_local
                                2,22% ThreadStateTransition::transition_and_fence
                                1,65% __pthread_getspecific
                             0,67% JNIHandles::make_local
                             0,53% JvmtiVMObjectAllocEventCollector::~JvmtiVMObjectAllocEventCollector
                     + 0,86% Interpreter
      + 2,86% GCTaskThread::run
+   98,78%     0,00%  java  libjvm.so           [.] java_start
+   95,77%     0,00%  java  libjvm.so           [.] JavaThread::run
+   95,77%     0,00%  java  libjvm.so           [.] JavaThread::thread_main_inner
+   95,54%     0,02%  java  perf-20431.map      [.] Interpreter
+   95,54%     0,00%  java  libjvm.so           [.] thread_entry
+   95,54%     0,00%  java  libjvm.so           [.] JavaCalls::call_virtual
+   95,54%     0,00%  java  libjvm.so           [.] JavaCalls::call_virtual
+   95,54%     0,00%  java  libjvm.so           [.] JavaCalls::call_helper
```

Hardware-based Approach



External

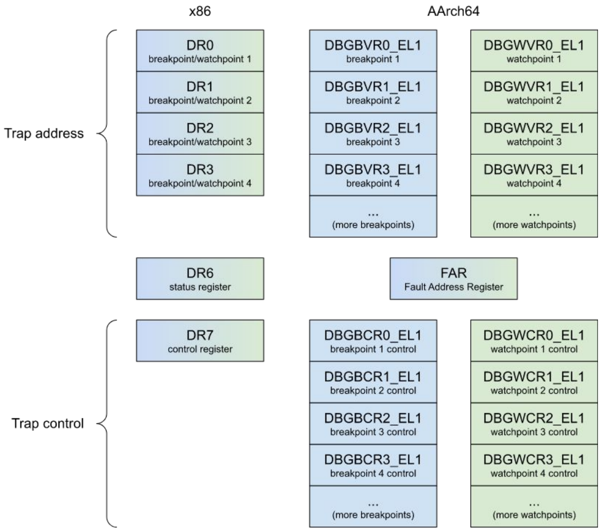
- Arm CoreSight

Internal

- Hardware watchpoints/breakpoints
  - Very few usually available - four in x86!
  - Can detect loads, stores, and fetches
- Intel Branch Trace Store (BTS)
  - 40x application runtime slowdown
  - Deprecated
- Intel Processor Trace (PT)
  - Ah - interesting!
  - <5% slowdown

Fast!

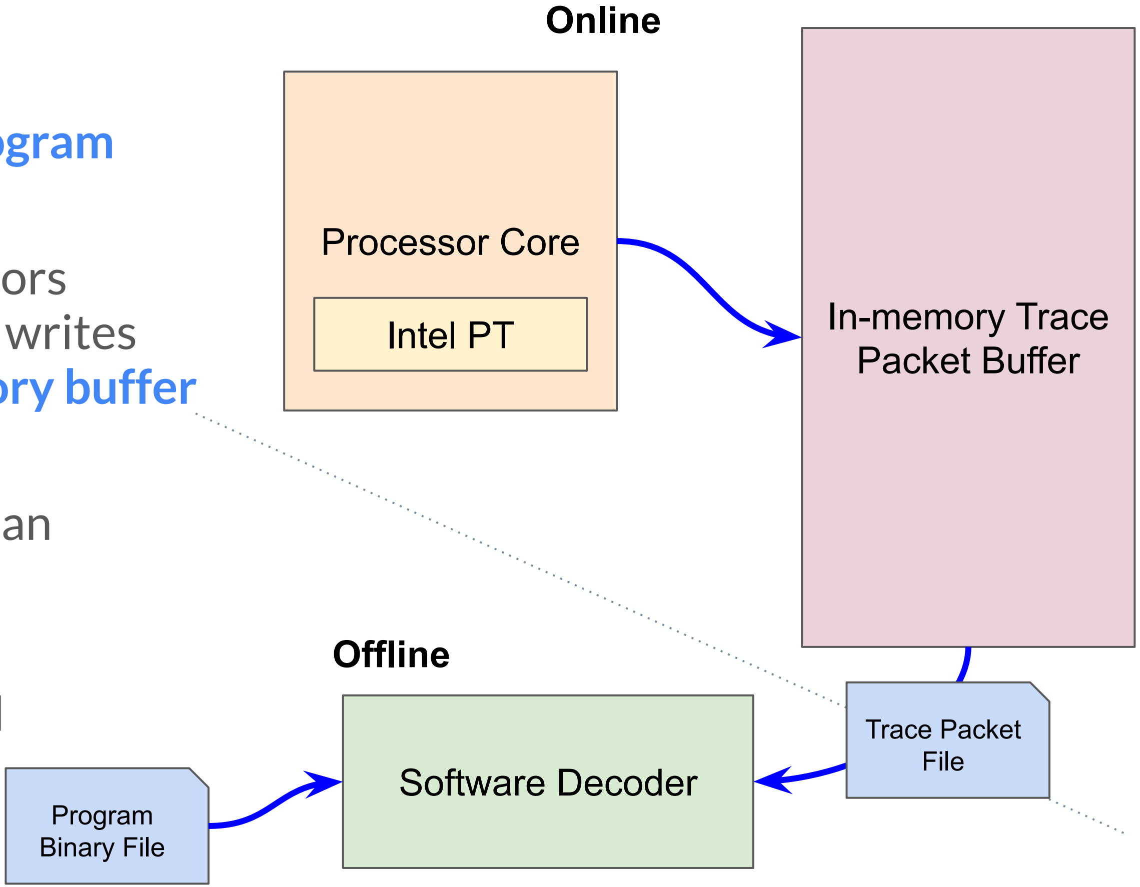
Too fast!



	x86	AArch64	
Trap address	DR0 breakpoint/watchpoint 1	DBGBVR0_EL1 breakpoint 1	DBGWVR0_EL1 watchpoint 1
	DR1 breakpoint/watchpoint 2	DBGBVR1_EL1 breakpoint 2	DBGWVR1_EL1 watchpoint 2
	DR2 breakpoint/watchpoint 3	DBGBVR2_EL1 breakpoint 3	DBGWVR2_EL1 watchpoint 3
	DR3 breakpoint/watchpoint 4	DBGBVR3_EL1 breakpoint 4	DBGWVR3_EL1 watchpoint 4
		... (more breakpoints)	... (more watchpoints)
	DR6 status register	FAR Fault Address Register	
Trap control	DR7 control register	DBGBCR0_EL1 breakpoint 1 control	DBGWCR0_EL1 watchpoint 1 control
		DBGBCR1_EL1 breakpoint 2 control	DBGWCR1_EL1 watchpoint 2 control
		DBGBCR2_EL1 breakpoint 3 control	DBGWCR2_EL1 watchpoint 3 control
		DBGBCR3_EL1 breakpoint 4 control	DBGWCR3_EL1 watchpoint 4 control
		... (more breakpoints)	... (more watchpoints)

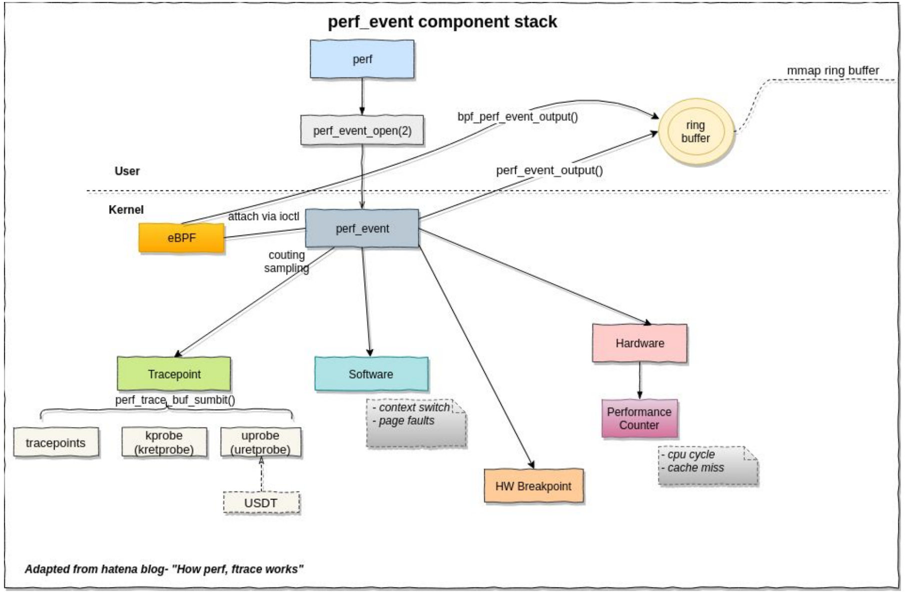
Intel Processor Trace

- **Hardware accelerated program execution tracing**
- **Online:** "**Externally**" monitors execution of software, and writes tracing data to an **in-memory buffer**
- **Offline:** Using the original source-code and compiler, an execution trace can be **reconstructed**
- **Very little** online overhead



Intel Processor Trace

- OS support **required**
  - `perf`
  - `simple-pt` (Andi Kleen, Intel)
- Same-architecture tracing (x86)
- **Does not generate** information about unconditional direct branches
- **Generates** only result of conditional direct branches
- **Generates** target address for indirect branches
- Highly **compressed** packet representation
- JITted code **not “supported”**
  - JIT must generate additional information

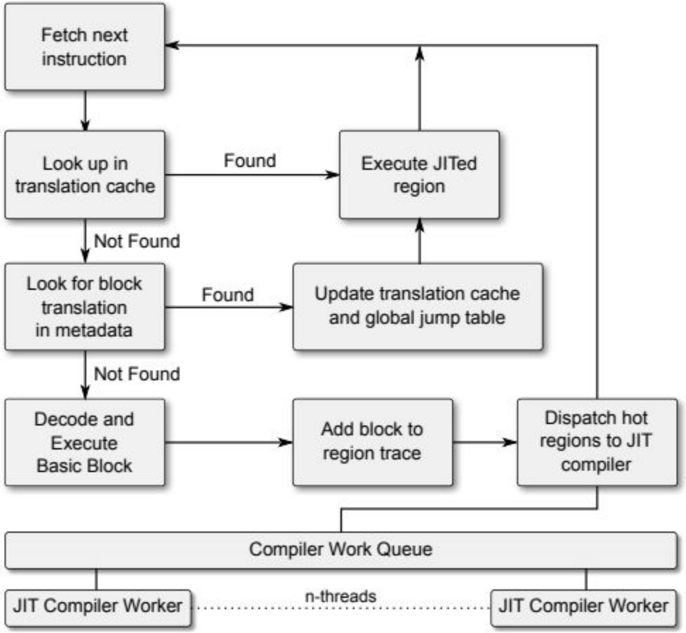


Adapted from hatena blog- "How perf, ftrace works"

Dynamic Binary Translation

- **Same-architecture DBT**
  - Instrumentation, e.g. Intel PIN
- **Cross-architecture DBT**
  - Instruction Set Simulators
    - Qemu
    - ArchSim
    - Captive
  - Legacy application execution
    - Apple Rosetta
  - Normally implemented with **Just-in-time Compilation**

This is what I'm interested in!


Fetch next instruction
Look up in translation cache
Found
Execute JITed region
Not Found
Look for block translation in metadata
Found
Update translation cache and global jump table
Not Found
Decode and Execute Basic Block
Add block to region trace
Dispatch hot regions to JIT compiler
Compiler Work Queue
JIT Compiler Worker
n-threads
JIT Compiler Worker

Terminology

ISA

Instruction Set Architecture

Host

The ISA on which the translation runtime is executing, e.g. x86

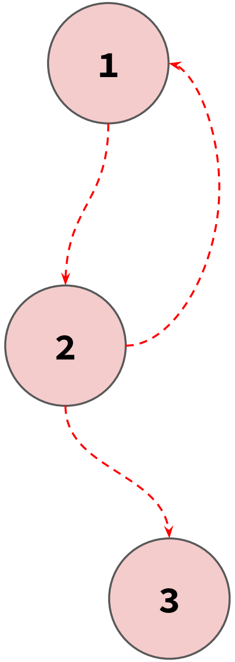
Guest

The ISA which is being executed using DBT, e.g. Arm

Debugging

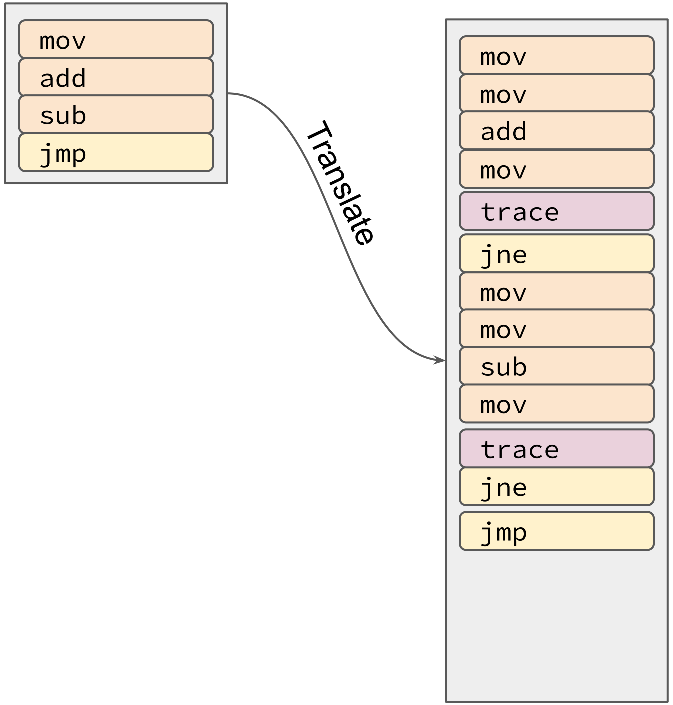
Tricky!

- Guest program has its own **effective control-flow**
  - i.e. what would be observed if it was running natively.
- Host machine is executing **translated code**, which probably doesn't correspond to guest code.
- How to collect **guest execution trace**?



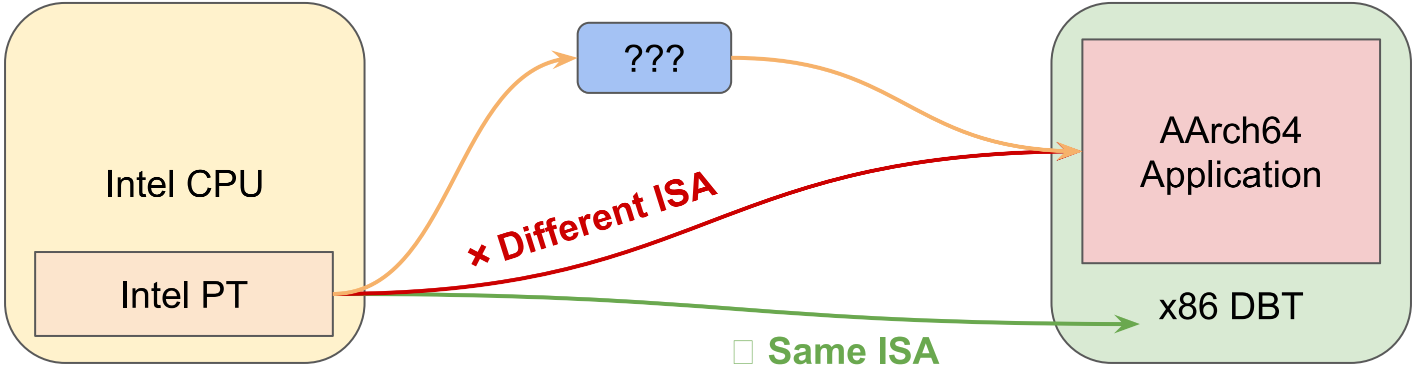
Software Tracing

- It’s a DBT, so use **instrumentation**!
- Use an **external** tracing tool, e.g. `perf`
  - But what corresponds to what?
- **Slow!** We’re back to generic software tracing...



Hardware Tracing

- No **built-in support** for Dynamic Binary Translation
- Extra work required to support **JIT compiled code**
- **Can we exploit it for what we want to do?**



???

Intel CPU

Intel PT

× Different ISA

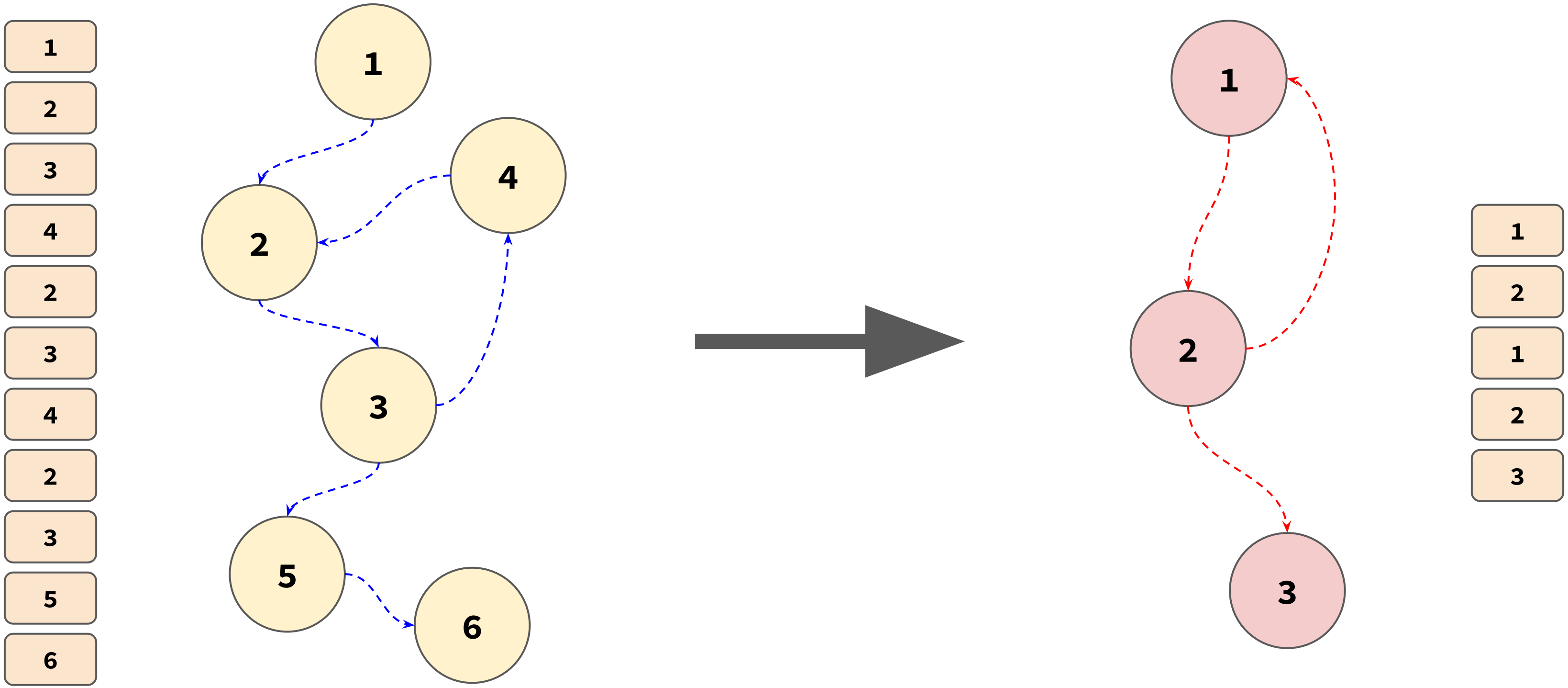
AArch64 Application

x86 DBT

□ Same ISA

Hardware Tracing

Idea: Collect native host trace, and map it to an equivalent guest trace

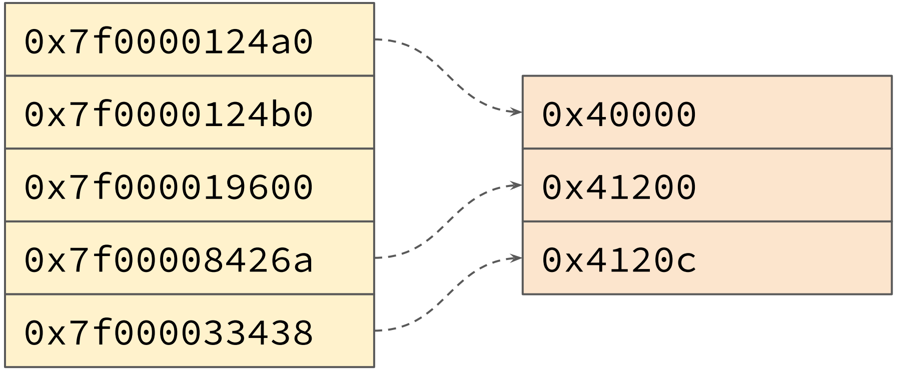


Mapping

- **Map host basic-blocks to guest basic-blocks**
- Host basic-blocks are generated by executing guest basic-blocks

But... Host basic-blocks may have more or less control-flow than guest!

- Runtime **dynamic** control-flow
  - e.g. control-flow within an instruction emulation
- Translated code **optimisation**
  - e.g. elimination of branches due to trace-based compilation



0x7f0000124a0

0x7f0000124b0

0x7f000019600

0x7f00008426a

0x7f00033438

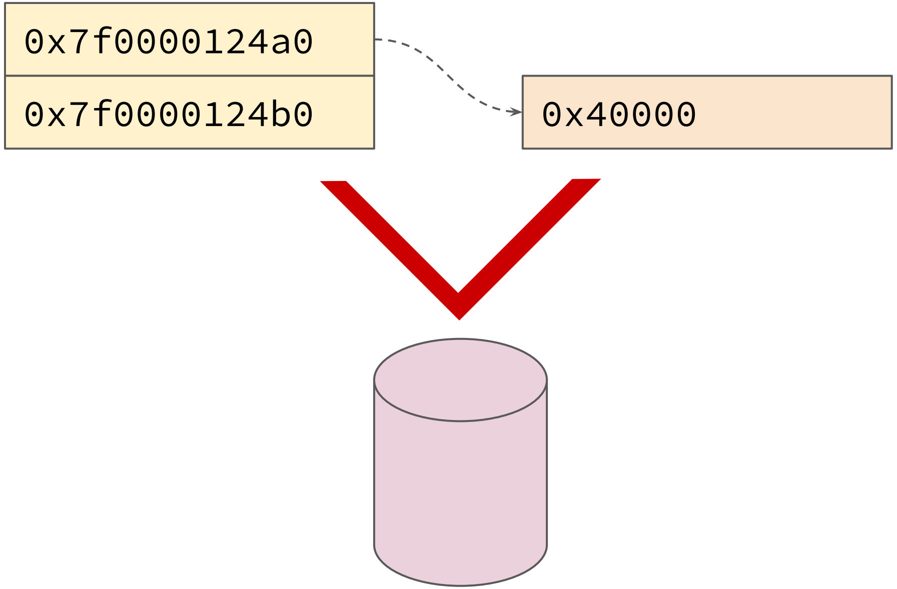
0x40000

0x41200

0x4120c

Challenges

- How do we **efficiently** produce this mapping?
  - What if mappings change? e.g. DBT recompilation
- Intel PT produces tracing information **TOO QUICKLY**
  - **No chance** of any online decoding
  - **Any chance** of collecting a perfect trace?
  - Storage volume/bandwidth requirements **huge**!
- Need to consider time for **offline processing**, vs a **software implementation**



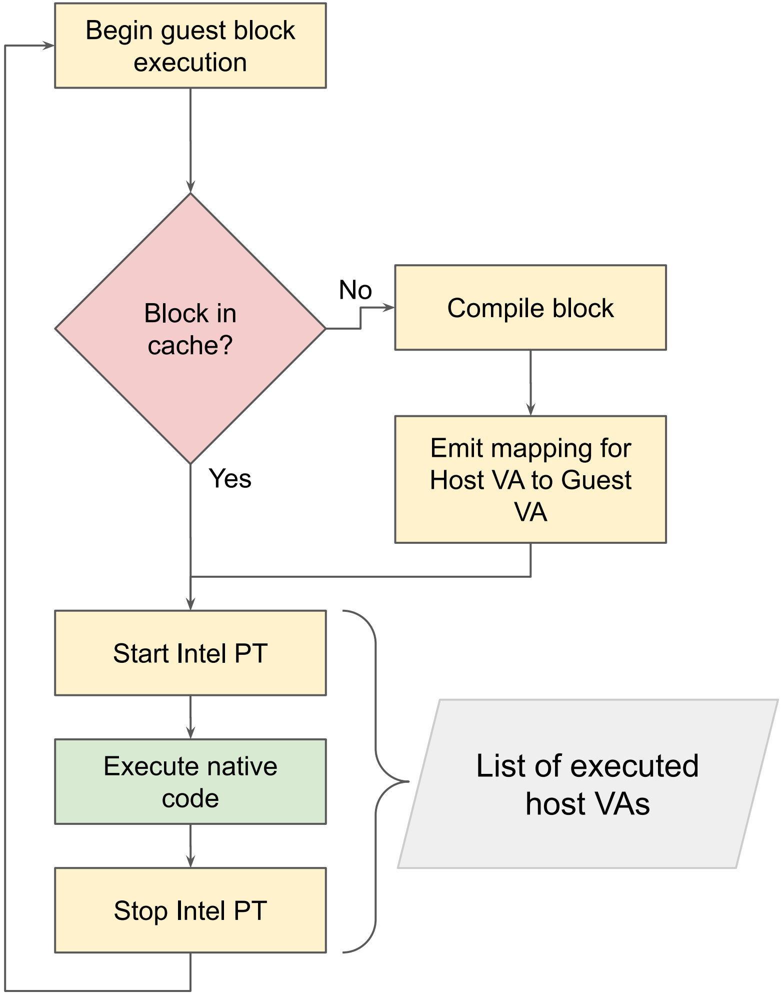
0x7f0000124a0

0x7f0000124b0

0x40000

Proof-of-concept

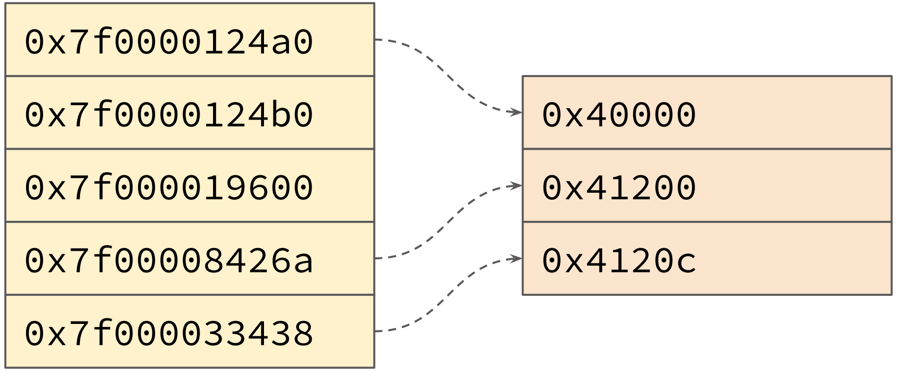
- **Implemented in Qemu**
- x86 host machine, AArch64 guest machine
- Qemu **enables** Intel PT on entry into translated code
- Intel PT trace written to file on disk
- Qemu **disables** Intel PT on exit from translated code
- Block chaining **keeps** Qemu in translated code
- Map file generated containing **host virtual addresses** of translated code representing **guest virtual addresses**



Begin guest block execution
Block in cache?
No
Compile block
Emit mapping for Host VA to Guest VA
Yes
Start Intel PT
Execute native code
Stop Intel PT
List of executed host VAs

Proof-of-concept

- Intel PT trace is **decoded** into list of **host virtual addresses**.
- For each host VA, map file is consulted to see if **corresponding guest virtual address** exists.
- If there's a **match**, the **guest VA** is written to the output trace.
- If there **isn't a match** - it's ignored.



0x7f0000124a0

0x7f0000124b0

0x7f000019600

0x7f00008426a

0x7f000033438

0x40000

0x41200

0x4120c

Proof-of-concept

Possibility of significant speed-up!

No Tracing: 24.607s

Naive Tracing: 149.21s

PT External: Perf: 31.55s

PT Internal: No Chain: 7780.21s

PT Internal: Indirect Chain: 25.41s

What’s next

Software domain:

- Talk directly to Intel PT
  - Custom kernel driver
- Artificially slow down execution of guest
  - Adaptive rate control

Hardware domain:

- Hardware unit for processing and consuming trace data
  - DMA directly from PT trace buffer into “translated” trace

Thank-you!

Questions?

Tom Spink

tcs6@st-andrews.ac.uk

https://tcs6.host.cs.st-andrews.ac.uk