

Introduction to Cavatools

Background

There is an unfortunate tension that must be dealt with when designing high-performance computer systems: the desired system is extremely fast, but the simulation means available to exercise design options, compare, and decide are very, very slow.

A typical example is the gem5 simulator; the following table¹ shows that for a particular configuration of the gem5 simulator running some benchmarks on a host system using 1-8 cores, execution of simulated instructions is less than 200 thousand instructions per second (KIPS)

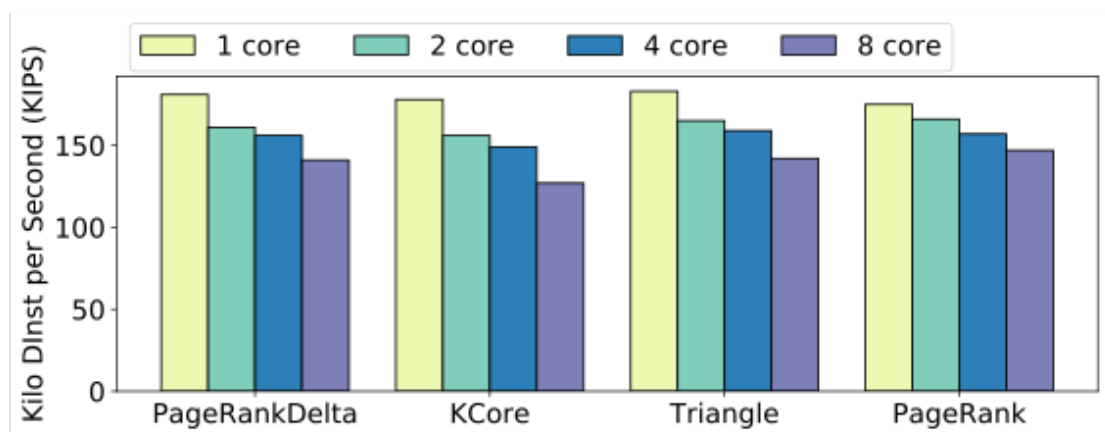


Figure 2: Performance of gem5 in Multi-Core Simulations

Another figure in the same paper shows how fast gem5 runs when doing “fast-forwarding” - essentially acting as an interpreter to provide quick functional (rather than timing-accurate) simulation:

¹ Tuan Ta, Lin Cheng, and Christopher Batten, “Simulating Multi-Core RISC-V Systems in gem5”, 2018, <https://www.csl.cornell.edu/~cbatten/pdfs/ta-gem5-riscv-carrv2018.pdf>

| Benchmarks | DInst-FF (M) | DInst-Detailed (M) | Speedup |
|---------------|--------------|--------------------|---------|
| PageRank | 496 | 2858 | 1.16x |
| PageRankDelta | 496 | 400 | 2.01x |
| KCore | 496 | 628 | 1.67x |
| Triangle | 496 | 1069 | 1.42x |

Table 8: Performance Speedup over Full Simulations without Fast-Forwarding Mode – DInst-FF = number of dynamic instructions that are fast-forwarded. DInst-Detailed = number of dynamic instructions that are simulated in the detailed mode.

The speedup is only around a factor of 2X, so simulation rates don't exceed 300KIPS or so. When your target system is a multiprocessor, and each core runs in the gigahertz range, the performance gap between simulation and reality is extremely large — a few billion instructions per second is several thousand times faster than 300 KIPS.

This vast performance gulf leads to major problems when we want to exercise our target design with realistic large programs.

Cavatoools is a new approach to system simulation which seeks to offer 100X performance improvement, in the 100-200 million instructions per second (MIPS) range. It does this using two main ideas: (1) break problem of modeling a single processor core into a pipeline of operations that can be simulated in parallel on a host computer with multiple cores, and (2) take advantage of RISC-V Weak Memory Ordering to model multiple cores as parallel (pipeline of) processes running on host computer with (more) parallel cores, synchronizing only when the simulated program performs memory operations that have defined system-wide ordering, such as LR and SC, atomic memory operations (AMO), and the FENCE instruction.

The head of the modeling pipeline is an interpreter which produces an execution trace of program counter (PC) and load/store memory addresses in shared memory. The second element in the pipeline is a functional simulator which produces another trace in shared memory. This functional simulator models the processor instruction decode and execution pipeline, and a minimal subset of the cache hierarchy whose activity can be overlapped with the processor pipeline on a cycle-by-cycle basis. For in-order processors this typically includes the instruction buffer and the first level data cache. The functional simulator outputs a trace containing instruction issue timing and data cache misses.

The next elements in the pipeline consists of one or more cache simulators, terminating in a main memory simulator (which may be integrated into the last cache simulator). The cache

simulators model the L1 instruction cache, the L2 caches (may be unified or separate instruction and data caches simulated by two cache simulators), etc.

Cavatoools has a universal trace record format allowing any number of cache simulators to be chained together to model memory hierarchy of L1, L2, L3... There is a funnel operator for merging multiple cache miss traces to create a composite output trace with correct timing. The merged trace can be fed into another cache simulator to model, for example, a shared L2 cache.

A multi-core system is modeled as a tree of cache simulators with a memory controller model at the root and pipeline simulators at the leaves, each fed by its own copy of the RISC-V instruction interpreter. All the nodes of this modeling tree are independent Linux processes and run in parallel on a multi-core host computer; they can be controlled using standard Linux thread-to-core affinity utilities.

The following not yet operational. When the interpreter encountered a synchronization instructions (memory operation with defined strong global ordering), it must wait until the appropriate “global” time to make the store visible. There is concept of a “global clock” which represents the oldest clock cycle of any functional simulator. The global clock is advances periodically by whichever is the trailing simulator of the moment as different functional simulators make forward progress at different rate. There is no set requirement on how frequently the global clock must be updated—too frequently will cause excessive synchronization overhead, too infrequently will impede progress of simulated program when it performs synchronizing (ordered) memory operations.

The simulated multi-core machine correctly models Weak Memory Ordering. Because the timing of simulated synchronization events is determined by real hardware events happening on host processor cores which do not have deterministic execution time (because of cache misses, asynchronous interrupts, thermal throttling, etc.), the simulation cannot deliver cycle-accurate repeatable results—it is analogous to the behavior of a real multi-core machine running a parallel program.

The RISC-V version of Cavatoools is a further development of the cava toolset developed by Peter Hsu 1990-2005. Contributions from Pete Wilson and Borja Perez have been incorporated along with further work by Peter Hsu. At the time of writing all are at the Barcelona Supercomputing Centre.

Cavatoools is open-source software licensed under the Apache license. See file LICENSE.

RISC-V Interpreter

The instruction set interpreter is called **caveat**.

Usage: `caveat [caveat-options] target-program [target-options]`

Options: `[choice, default (1st) value]`

```

--out=          Create trace file/fifo =name [no trace]
--trace=       synonym for --out
--buffer=      Shared memory buffer size is 2^ =n bytes [12]
--func=        Trace function =name [_start]
--withregs     Include register values in trace
--after=       Start tracing function after =number calls [1]
--every=       Trace only every =number times function is called [1]
--report=      Progress report every =number million instructions [1000]
--quiet        Don't report progress to stderr
-q            short for --quiet

```

At this time only statically linked binaries are supported. Both `libc` and `newlib` binaries, compiled using toolchains `riscv64-unknown-elf-cc` and `riscv64-unknown-linux-gnu-cc` (plus other languages), respectively, are supported.

Without options `caveat` simply runs the program—it behaves as a user-mode Linux RISC-V virtual machine. By default `caveat` prints performance status periodically to `stderr`. The frequency of status reports can be changed to every `--report=n` instructions, or turned off using the `--quiet` flag.

For performance analysis `--out=name` tells `caveat` to generate a trace. Currently traces are dynamic FIFO objects in shared memory, to be consumed by another process on the same computer. In future other options will be available. The trace FIFO object is created as `/dev/shm/name` in the file system and `name` must be unique for each concurrently running `caveat` process. The shared memory buffer size is optionally specified by `--buffer=n` to be 2^n bytes.

By default the entire program is traced. A specific function or subroutine can be traced using the `--func=name` option. For C++ programs with mangled names, only the leading part of the function name must match.

By default every invocation of the function is traced. The `--after=n` option skips the first `n` calls. The `--after=n` option traces every `n`'th time the function is called. These options can be used together to statistically sample a function in real applications that run for a long time.

The `--withregs` option includes register update values to be included in the trace. This enables an analysis program to recreate the complete execution state of the program. This feature is useful to verify downstream simulators.

Pipeline Simulator

Cavatoools comes with a single-issue in-order pipeline simulator called ***pipesim***.

Usage: `pipesim --in=trace [pipesim-options] target-program`

```
Options: [choice, default (1st) value]
--in=          Trace file from caveat =name
--bdelay=     Taken branch delay is =number cycles [2]
--imiss=      L0 instruction buffer refill latency is =number cycles [5]
--iline=      L0 instruction buffer line size is 2^ =n bytes [8]
--iblsz=      L0 instruction buffer block size is 2^ =n bytes [4]
--dmiss=      L1 data cache miss latency is =number cycles [25]
--write=      L1 data cache is write=[back|thru]
--dline=      L1 data cache line size is 2^ =n bytes [6]
--dways=      L1 data cache is =w ways set associativity [4]
--dsets=      L1 data cache has 2^ =n sets per way [6]
--out=        Create output trace file =name [no output trace]
--timing       Include pipeline timing information in trace
--report=     Progress report every =number million instructions [100]
--quiet       Don't report progress to stderr
-q           short for --quiet
```

In addition to the instruction fetch, decode and execution pipeline, *pipesim* models a simple instruction buffer and a first-level data cache. Instruction buffer fetch misses and L1 data cache misses are counted. If the optionally `--out=name` parameter is given the instruction buffer misses and L1 data cache misses are traced for further processing. Instruction execution timing information can be included in the trace file for further analysis by adding the `--timing` option.

The instruction buffer model is precisely characterized as a two-way set-associative, one-set cache with long sub-blocked cache lines. It consists of two cache lines, a MRU (most recently used) and a LRU (least recently used) line. The replacement buffer is very simple: if hit MRU, do nothing. If hit LRU, exchange MRU, LRU status (flip bit). If miss both, MRU becomes LRU, refill MRU with new line. The cache lines are sub-blocked. The critical sub-block is filled first, then sequential sub-blocks forward are filled, wrapping around.

The parameter `--iline=n` specifies the instruction cache line size is 2^n bytes. The parameter `--iblsz=n` specifies the block size is 2^n bytes. The refill latency or miss penalty is given by `--imiss=c` cycles. Taken branches flush the pipeline, incurring a delay of `--bdelay=n` cycles.⁷

This simple type of instruction buffer has been used effectively for scientific computers since the 1950's. The Cray-1 had a 4-line instruction buffer of this type—in the 1970's compilers were less advanced and it was necessary to have two cache lines for DAXPY (because the subroutine may span cache line boundaries), and two more cache lines for the enclosing loop that called DAXPY. Nowadays compilers would inline the subroutine, so we need only two longer cache lines.⁹

The first level data cache in *pipesim* can be configured as `--write=back` or `--write=thru`, the default is write-back. The parameter `--dline=n` specifies the data

cache line size is 2^n bytes, `--dsets=n` specifies 2^n sets, `--dways=n` specifies set-associativity ($1 \leq n \leq 4$).

The data cache refill latency or miss penalty is `--dmiss=c` cycles. Pipesim has a non-blocking data cache miss model: when a load instruction causes a cache miss, the result register is marked busy and the pipeline stalls only when a subsequent instruction uses the value in that register. The number of stall cycles depend on how long ago the cache miss was launched. This allows software to schedule instructions to overlap cache misses.

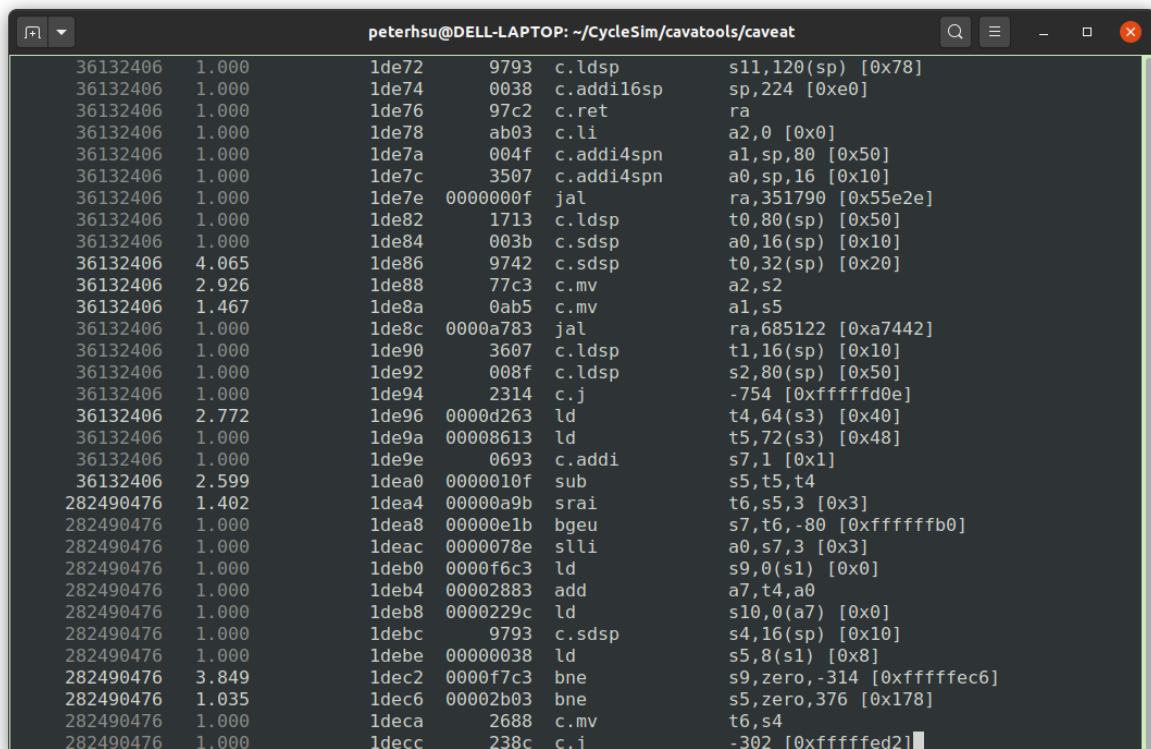
Cavatoos comes with Python script ***softpipe*** that create source-level software pipelines of simple loops to demonstrate cache miss overlapping.

The data cache is virtually indexed and virtually tagged; there is no TLB in pipesim. This may be adequate for accelerator designs, but may have to be revisited in the future.

Realtime Viewer

It is possible to observe *pipesim* as it executes a program in realtime using ***erised***. Run the following in different windows, where *interesting* is function of interest in *test-program*.

```
$ caveat --out=tracel --func=interesting target-program
$ pipesim --in=tracel --count=count1 target-program
$ erised --count=count1 --func=interesting target-program
```



```
peterhsu@DELL-LAPTOP: ~/CycleSim/cavatoos/caveat
36132406 1.000 1de72 9793 c.ldsp s11,120(sp) [0x78]
36132406 1.000 1de74 0038 c.addi16sp sp,224 [0xe0]
36132406 1.000 1de76 97c2 c.ret ra
36132406 1.000 1de78 ab03 c.li a2,0 [0x0]
36132406 1.000 1de7a 004f c.addi4spn a1,sp,80 [0x50]
36132406 1.000 1de7c 3507 c.addi4spn a0,sp,16 [0x10]
36132406 1.000 1de7e 0000000f jal ra,351790 [0x55e2e]
36132406 1.000 1de82 1713 c.ldsp t0,80(sp) [0x50]
36132406 1.000 1de84 003b c.sdsp a0,16(sp) [0x10]
36132406 4.065 1de86 9742 c.sdsp t0,32(sp) [0x20]
36132406 2.926 1de88 77c3 c.mv a2,s2
36132406 1.467 1de8a 0ab5 c.mv a1,s5
36132406 1.000 1de8c 0000a783 jal ra,685122 [0xa7442]
36132406 1.000 1de90 3607 c.ldsp t1,16(sp) [0x10]
36132406 1.000 1de92 008f c.ldsp s2,80(sp) [0x50]
36132406 1.000 1de94 2314 c.j -754 [0xfffffd0e]
36132406 2.772 1de96 0000d263 ld t4,64(s3) [0x40]
36132406 1.000 1de9a 00008613 ld t5,72(s3) [0x48]
36132406 1.000 1de9e 0693 c.addi s7,1 [0x1]
36132406 2.599 1dea0 0000010f sub s5,t5,t4
282490476 1.402 1dea4 00000a9b srai t6,s5,3 [0x3]
282490476 1.000 1dea8 00000e1b bgeu s7,t6,-80 [0xfffffb0]
282490476 1.000 1deac 0000078e slli a0,s7,3 [0x3]
282490476 1.000 1deb0 0000f6c3 ld s9,0(s1) [0x0]
282490476 1.000 1deb4 00002883 add a7,t4,a0
282490476 1.000 1deb8 0000229c ld s10,0(a7) [0x0]
282490476 1.000 1debc 9793 c.sdsp s4,16(sp) [0x10]
282490476 1.000 1debe 00000038 ld s5,8(s1) [0x8]
282490476 3.849 1dec2 0000f7c3 bne s9,zero,-314 [0xfffffec6]
282490476 1.035 1dec6 00002b03 bne s5,zero,376 [0x178]
282490476 1.000 1deca 2688 c.mv t6,s4
282490476 1.000 1decc 238c c.j -302 [0xfffffd2]
```

The last window will show assembly listing of function as shown in the figure. Use up and down arrow keys or mouse wheel to scroll through function. If target-program is long running the display will be in real time. Otherwise it will display the final counts.

The first column is number of times the instruction has executed. The second column is the average number of cycles that instruction took to issue, the CPI for that instruction. The rest of the line is disassembly of the instruction: PC, hex of the instruction bits, etc.

If the CPI is very close to 1.0 the counts are displayed dimly. This makes it easy to see where the stalls are.

Cache Simulator

A generic cache simulator **cachesim** is included with Cavatools, algorithm provided by Peter Wilson.

```
Usage: cachesim --in=trace [cachesim-options]
      --out=x can be another cachesim --in=x for multilevel simulation
Options: [choice, default (1st) value]
--in=          Trace file =name (from caveat, pipesim, or cachesim)
--line=        Cache line size is 2^ =n bytes [6]
--ways=        Cache is =w ways set associativity [8]
--sets=        Cache has 2^ =n sets per way [11]
--sim=         Simulate all access =types [iIdD0123rRwW] default all
--out=         Output next-level misses to trace =name [no next level]
--report=      Progress report every =number million instructions [10]
--quiet        Don't report progress to stderr
-q            short for --quiet
```

The parameter names are similar to *pipesim*: `--line=n` for log-base-2 line size in bytes, `--sets=n` for log-base-2 number of sets, and `--ways=n` for number of ways set-associativity. The usual reporting frequency and quiet options are available. Status is written to *stderr*, final statistics is written to *stdout*.

The `--sim=chars` character string option specifies the type of cache to simulate. The cache can be an instruction cache (characters 'i' or 'I'), a data cache ('d' or 'D'), or both (multiple characters 'id' or 'ID'). It can be a read-only cache ('r' or 'R'), a write-only cache (i.e. write buffer, 'w' or 'W'), or read-write cache ('rw' or 'RW'). The level in the memory hierarchy is specified with a digit, '0', '1', '2' or '3'. A level-K cache simulates lower-level references as well, e.g. write through misses. The default is to simulate all memory references.

The `--in=name` option is mandatory. The `--out=name` option tells cachesim to write cache miss records to a trace for further processing. Incoming trace records that are not

processed (stores, for example, when the cache is configured as read-only) are passed through to the output trace for further processing (simulating a write-through cache).

The input and output trace formats are identical. Therefore multiple copies of *cachesim* can be concatenation to simulate a multilevel cache hierarchy. The trace format is compatible with both *caveat* and *pipesim*. Therefore *cachesim* can be used as a level-2 cache simulator for *pipesim* in a system model, or be driven by *caveat* directly to do cache miss-rate analysis without cycle times.

Trace Utility

There is a multipurpose program ***traceinfo*** that reads *caveat/pipesim/cachesim* traces.

```
Usage: traceinfo --in=trace [traceinfo-options] target-program
       - summarize trace, make listing or create derivative traces
Options: [choice, default (1st) value]
--in=          Trace file from any cavatools =name
--list         Print assembly listing (only traces from caveat)
--range=      Only interested in =begin,end addresses (Hex no 0x) [all]
--paraver=    Make Paraver trace of =cycles to stdout
--cutoff=     Ignore pipeline stalls less than =number cycles [1]
--report=     Progress report every =number million instructions [1000]
```

With no options (and no target-program) *traceinfo* simply counts trace events and prints a summary to *stdout*.

The `--list` option prints an assembly listing of the execution trace, including memory reference addresses. The trace must come directly from *caveat*. If the `--withregs` option was given to *caveat* then the listing will include register update values as well.

The `--paraver=n` option generates a trace to further processing and viewing using the BSC Paraver tool. The Paraver trace is limited to n cycles. Paraver events include instruction issue stalls and all levels of cache misses. Paraver traces can be derived from the output of *pipesim* (with `--timing` option) or after piping through one or more *cachesim*.

By default Paraver traces include all events (in the first n cycles). The `--range=begin,end` option omit events with program counter values outside of this range (inclusive). The addresses are given in Hex without leading 0x digits. Note the trace limit is given in cycles, not number of events.

The `--cutoff=n` option omit instruction issue stall events less than n cycles. By default all instruction issue stalls are recorded. Note instructions that issued without stalling are not recorded.

