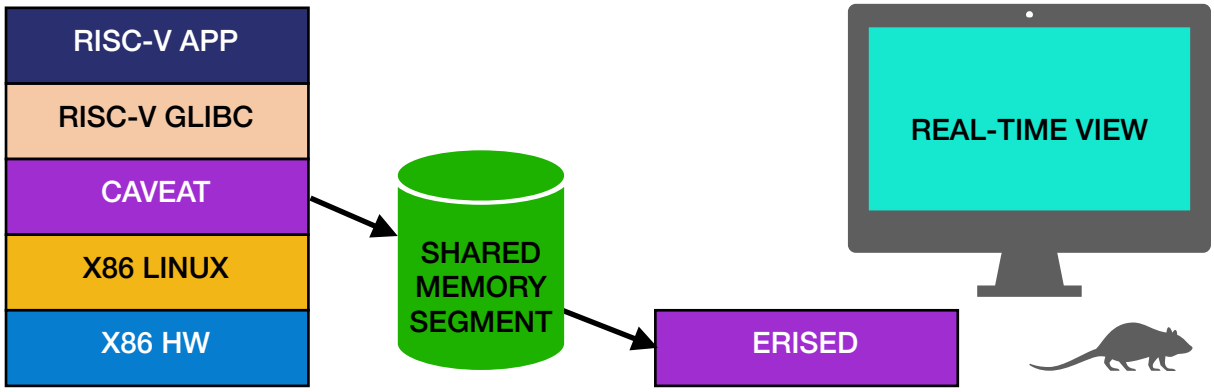
Cavatools Implementation

Peter Hsu Sep 16, 2021

Introduction

Many simulators can simulate a parallel machine running a parallel program. Not many simulators run the parallel simulation *in parallel* on a multiprocessor host computer. Cavatools simulates a multi-threaded virtual user-mode Linux RISC-V architecture on a multicore X86 Linux host computer. Linux facilities including all system calls from the host computer are available to the RISC-V virtual machine including fork/clone, mmap shared memory, asynchronous I/O and inter-process signal. The RISC-V guest program is compiled and linked using standard RISC-V tool chain with glibc and other standard libraries such as OpenMP, MPI, etc.

Cavatools has several parts: **Uspike** is a RISC-V interpreter. It presents a user-mode Linux virtual machine to the RISC-V guest program. **Caveat** is a performance simulator. At present it models a multicore processor with simple in-order instruction pipelines, private instruction and data level-1 caches, a unified shared level-2 cache with snoopy bus, and a simplistic fixed latency main memory. **Erised** is a real-time viewer. Caveat creates a shared memory segment with real-time performance counters for each program counter location: the number of times that instruction was executed, the number of cycles attributed to this instruction including dependency stalls, cache misses and pipeline draining due to branches. Erised shows this detailed information at 30 frames per second while the simulation is running—the mouse is used to zoom in and scroll around program regions of interest.



The interpreter part of uspike is a library. The instruction execution semantics is mechanically derived from the RISC-V “golden simulator” Spike. Uspike understands all user-mode instructions supported by Spike including the Vector Extension “V” and various proposed extensions such as Bit Manipulation “B” and Packed-SIMD “P”. The intention is to mechanically track Spike to avoid errors as new instruction extensions are created.

Caveat is linked with the uspike library to obtain instruction execution semantics; it models timing and does not need to be concerned with interpretation semantics. Caveat is a C++ program designed to be easily extensible to model a variety of processor pipelines and cache organization. The simulator runs in situ with the interpreter; no instruction trace is created. Caveat is designed to be a fast simulator capable of running large productions programs. We have successfully simulated GROMACS running on multiple simulated RISC-V cores. GROMACS is a million-line molecular modeling application and major workload on the Mare Nostrum Supercomputer at Barcelona Supercomputing Center in Spain.

This paper discusses implementation details of Cavatools. We do not claim novel inventions; all the technologies discussed here have likely been implemented in one form or another by other virtual machines and simulators. However most virtual machines and simulators are proprietary software of commercial companies and detailed information on algorithms and implementation have not been published. Cavatools is open source software.

Virtual RISC-V Linux Machine

Uspike presents a user-mode Linux to the guest RISC-V program. It is an interpreter running on X86 Linux. The RISC-V guest program is compiled using standard RISC-V compilers (GCC and LLVM) and linked with the standard glibc library. Uspike emulates RISC-V instructions using semantics derived from the RISC-V "golden simulator" Spike: Python scripts extract instruction encoding bit patterns and C++ execution semantics code directly from the riscv-isa-sim and riscv-tools repositories. The intention is to minimize human translation errors, and to continuously track Spike as RISC-V instruction set extensions evolve.

Uspike loads the RISC-V guest program in the host address space at the position the guest program is linked for. Therefore loads and stores of static and global variables, as well as PC locations for branches, calls and returns, refer to linked addresses exactly as if the program ran natively on real RISC-V hardware. The interpreter/simulator itself is "hidden away" at high address above the stack.

There are two reasons we use this technique. One reason is load/store emulation speed. A RISC-V load word instruction `lw` with address register plus immediate is interpreted by C++ statement `*(int32_t*)(xreg[rs1]+imm)` where the fields `rs1` and `imm` are pre-decoded values from the instruction and `xreg[]` is array of integer register values. Address translation is performed by the host X86 processor using its hardware TLB. Loads and stores typically constitute some 30% of all dynamic instruction instances; emulation speed of loads and stores have a big impact on overall interpretation speed.

The second reason is ease and speed of system calls. Uspike proxy RISC-V system calls by passing them to the underlying X86 Linux. System call parameters are typically passed in registers, but the registers may contain pointers to structures. The pointed-to data structures may themselves contain pointers to other data or function addresses. Both RISC-V and X86 are Little Endian architectures. When the guest program is loaded at the position the program is linked for, we can proxy the RISC-V system call by simply passing it through to the underlying X86 Linux. The X86 Linux read and write RISC-V program data through pointers just as if it was a native X86 program. There is basically no overhead to proxy system calls.

If the interpreted program is loaded in a different place in memory such as in a "memory array," then system calls must be individually proxied to copy and relocate all embedded pointers in any data structures used by the system call. Since there are a great many Linux system calls, and since many of them reference different structures with pointers depending on flag or option bits in the argument, full support for all Linux calls is both laborious and error prone. For example Spike's Proxy Kernel PK only supports the most common system calls and cannot run large production programs such as GROMACS.

Heap Management

The interpreter is a C++ program linked with glibc. It contains calls to new and delete operators which in turn calls malloc() and free(). The standard malloc uses the brk() system call to allocate memory. The guest program has been linked with the guest glibc which contains its own copy of malloc() and free() in the guest ISA. That malloc also uses system call brk() to allocate memory. The glibc malloc algorithm is very sophisticated and caches a lot of state. It is essential to avoid confusion between the host interpreter malloc and the guest malloc.

We provide our own version of malloc() and free() in the interpreter as well as C++ operators new and delete. The interpreter malloc() is a simple algorithm that uses a memory region obtained through mmap() and never calls brk(). Therefore the interpreter's heap becomes the guest program's heap when the interpreter makes system call brk() on behalf of the guest program. Note the guest program's heap in interpretation may begin at a different address than when the program runs natively under RISC-V Linux. However proper C and C++ program are supposed to use system call brk(0) to find the beginning address of the heap, which may be different run-to-run for security reasons. Also for security reasons the stack may begin at a different address run-to-run (called Stack Randomization).

Thread Cloning

When the guest RISC-V program executes the clone() system call we must create another virtual processor for the new thread. This is done with the clonecpu() method, which allocates a new Spike processor_t class then copies the current register files and control register state to the new cpu_t structure. We also allocate a new stack (and thread-local storage if necessary) for the new interpreter thread. Note the guest program has also allocated a new stack and thread-local storage before the clone() system call. The interpreter's new stack is different from the guest program's new stack—the child interpreter thread runs the new guest thread using the new interpreter stack, while the new guest thread uses the stack provided in the clone() system call.

The Linux clone() system call can be used to create a new thread in the same address space, as well as a thread in a new address space. The latter was previously called fork() but is now implemented as an option to clone(). To correctly proxy the fork() behavior it is necessary to allocate the new interpreter state in the *new* address space.

Linux fork() replicates the address space using copy-on-write page table entries. The child interpreter thread (which is a new Linux process in this case) must make a copy of the parent's processor state before the parent returns from fork() using a futex(). The child thread's malloc() will be in the new address space and Linux will provide new copy-on-write pages when stores are interpreted. However we must *not* chain the new processor to the list of processors because doing so will make the parent's processors appear in the child thread (process). Instead we make the child processor the first processor in the new process as if it was the thread running main(). We also proxy the execve() system call to load a new guest program for interpretation.

Time Management

Simulations of parallel processors must have the concept of time. Simulations of a single thread on a single processor can simplify time by using instruction sequence. But a parallel simulation must model the guest parallel program performing thread synchronization events. When a guest thread waits for an event from another guest thread, the simulator must track passage of time while the processor is stalled because processor utilization is one of the most interesting simulation results.

We will discuss hardware synchronization events later. Linux application programs use the futex() system call to perform software thread synchronization. Futex() can block the thread, causing Linux to context switch and run other threads. In the simulator each guest processor core is proxied by a simulator thread with its own interpreter. We want to be able to simulate many more processor cores than the number of real processor cores on the host computer. Therefore when a simulated thread calls futex() and blocks, the simulator thread should also block and context switch. This will automatically happen if we simply make the futex() system call on behalf of the guest thread. However we must track the passage of simulated time before and after the futex() call.

We have the concept of global_time and local_time. Local_time is the clock cycle counter of the processor core. Each simulated processor core has its own local_time and may be different from other processor core's local_time. Global_time is the system clock cycle.

Global_time can never be ahead of the slowest core: it is less than or equal to the local_time of every simulated processor core.

As each core is simulated its local_clock advances. How fast it advances depends on details of the simulated microarchitecture. For this discussion we will use the most simple microarchitecture: an IPC=1 machine where each and every instruction executes in one cycle. Each simulated core executes instructions and advances local_clock until it reaches a system call. While executing instructions the simulator thread periodically updates global_clock by traversing the list of processors to calculate the minimum local_clock value, then *atomically* updates global_clock if it is ahead. The atomic maximum function is necessary because multiple simulator threads may be computing min local_clock values simultaneously and trying to update global_clock independently. Global_time must advance monotonically and never get ahead of any local_clock; however it can can fall behind without harm.

System calls are modeled as events in global time. Before system calls are proxied, the simulator thread first compares global_clock to local_clock. If global_clock is behind local_clock the simulator thread issues a futex() and waits until global_clock is equal to local_clock. It then temporarily sets local_clock to +infinity and performs the system call on behalf of the guest. When the proxy system call returns, the simulator thread sets local_clock to the current global_clock, then continue simulation.

This algorithm makes guest program system calls “happen” in correct simulation time order. When one guest thread waits for a signal event from another guest thread, the sender advertises the sending time by waiting until global_clock equals local_clock. Note global_clock can never be greater than local_clock because it is computed as the minimum of all local_clocks. The waiting thread captures the signal event timing by setting its local_time to value of global_clock when it returns from the futex(). Global_clock is allowed to advance while the waiting thread is blocked because its local_clock is set to +infinity during the proxy system call.

The algorithm was developed for futex() in OpenMP applications. But it can be used to model timing of all system calls. In particular, it can be used to model Linux application threads that perform non-blocking read() and write() calls and uses signal() for synchronization. Many parallel cluster applications run across multiple Linux computers using socket() and non-blocking I/O to communicate. We believe it is possible to make caveat simulate parallel cluster applications, but this has not yet been verified.

Asynchronous Signals

Linux signal handling is initiated using the sigaction() system call to register a handler function which is asynchronously called when some event occurs. Events may be self-generated by the executing thread (eg. SEGV) or come from external sources such as completion of asynchronous I/O.

We proxy Linux signals by intercepting sigaction() system calls. When the guest application tries to register a signal handler function, the function pointer is for code in the guest ISA, so we cannot let Linux dispatch the host to that address. Instead the simulator records the guest signal handler function pointer and then registers a proxy signal handler using flag bits from the guest sigaction() call. When a signal is received the X86 Linux will invoke the proxy signal handler.

The proxy signal handler pretends to be Linux and creates an exception stack frame with the guest PC on the guest stack (using the RISC-V stack pointer register SP). Then it changes the guest PC to the saved guest signal handler and continues interpretation. The guest program may call sigaction() to change things, may call longjmp() to unwind its stack. The setjmp()/longjmp() facilities should “just work” without the interpreter’s help (not fully tested yet).

Note we do not change the local_time when receiving signals. The sending thread or process uses the kill() system to send the signal, or performed some I/O that the receiving

thread was asynchronously waiting for. Making any system call causes the sending thread to become synchronized to global_time. The receiving thread's local_time may be equal or ahead of global_time, but can never be behind. If local_time is equal to global_time, the signal is received "at the time it is sent." If local_time is ahead of global_time, the signal is delivered "a little bit later," with delivery latency being equal to difference between local_time and global_time. In either case it is "proper Linux signal behavior."

Cache Coherency and Atomic Memory Operations

Caveat simulates a group of RISC-V processor cores with private level-1 caches connected to a shared bus with a shared level-2 cache. Cache misses and atomic memory operations use the shared bus. RISC-V is defined with Weakly Ordered Memory Model. We use the time management scheme discussed above.

When a level-1 cache miss occurs or an instruction performs an atomic memory operation, we wait for global_time to catch up to the current processor's local_time. This waiting is performed using futex() system call to allow context switching by the host X86 computer so that it can simulate many more RISC-V cores than host X86 cores—RISC-V hardware cores are modeled as software threads on the host computer.

Once global_time has caught up with local_time, the interpreter thread sets its local_time to +infinity and arbitrates for the shared bus by acquiring a mutual exclusion lock. Time management is similar to system calls—after the interpreter acquires the mutex bus lock, it sets local_clock equal to global_clock and performs whatever level-1 cache coherency and level-2 cache operations, then sets global_time and its own local_time to however long the bus was occupied. Events on the shared bus and level-2 cache happen serially at global_clock time; if multiple interpreter threads arbitrate for the bus simultaneously the mutex cause them to be serviced one at a time with increasing global_time. The guest RISC-V thread "stalls" for the correct number of cycles.

Modeling shared bus and atomic memory operations is very performance critical. We expect a lot of tuning will be required to get good scaling on the host multicore computer.