Simulating Multi-Core RISC-V Systems in gem5

Tuan Ta, Lin Cheng, and Christopher Batten

School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{ qtt2, lc873, cbatten }@cornell.edu

ABSTRACT

The RISC-V ecosystem is becoming an increasingly popular option in both industry and academia. The ecosystem provides rich open-source software and hardware tool chains that enable computer architects to quickly leverage RISC-V in their research. While the RISC-V ecosystem includes functional-level, register-transfer-level, and FPGA simulation platforms, there is currently a lack of cycle-level simulation platforms for early design-space exploration. gem5 is a popular cycle-level simulation platform that provides reasonably flexible, fast, and accurate simulations. Previous work has added *single-core* RISC-V support to gem5. This paper presents our recent work on simulating *multi-core* RISC-V systems in gem5. We first describe our approach to functional and timing validation of RISC-V systems in gem5. We then evaluate the performance of the gem5/RISC-V simulator and discuss a design-space-exploration case study using gem5, the open-source RISC-V software tool chain, and two popular task-based parallel programming frameworks.

1 INTRODUCTION

RISC-V is an emerging open-source software and hardware ecosystem that has gained in popularity in both industry and academia [2, 11]. At the heart of the ecosystem, the RISC-V ISA is designed to be open, simple, extensible, and free to use. The RISC-V software tool chain includes open-source compilers (e.g., GNU/GCC and LLVM), a full Linux port, a GNU/GDB debugger, verification tools, and simulators. On the hardware side, several RISC-V prototypes (e.g., Celerity [4]) have been published. The rapid growth of the RISC-V ecosystem enables computer architects to quickly leverage RISC-V in their research.

Hardware modeling and simulation are critical for system design-space explorations. An ideal model is fast to simulate, accurate, and easy to modify. However, achieving all three goals in a single model is difficult (see Table 1). The RISC-V ecosystem provides functional-level models (e.g., Spike, QEMU), register-transfer-level (RTL) models (e.g., Rocket, Boom, Ariane), and FPGA models (e.g., Rocket Zedboard). Functional-level modeling is fast and easy to modify, but it does not capture the timing of the target system. RTL modeling provides cycle-accurate details of the target system at the cost of being slow to simulate and hard to modify. FPGA modeling provides both accurate and fast simulations but is even more challenging to modify owing to lengthy synthesis and place-and-route times. Cycle-level modeling offers a middle ground that is easier to modify than FPGA modeling, faster to simulate than RTL modeling, and more accurate than functional-level modeling. Its flexibility and performance provide a good platform for early system design-space exploration. We see a critical need for contributing open-source cycle-level models to the RISC-V ecosystem.

	Time to Modify	Time to Simulate	Accuracy
FL	+ +	+ +	− −
CL	+	+	−
RTL	−	− −	+ +
FPGA	− −	+ +	+ +

Table 1: Different Modeling Levels and Their Trade-Offs – FL = functional-level. CL = cycle-level. "+" and "-" show relative comparisons between levels.

gem5 is a popular cycle-level simulator that supports various instruction sets including x86, MIPS, and ARM. The simulator already provides a number of processor, cache, interconnection network, and DRAM models. It also offers advanced simulation features such as fast-forwarding and check-pointing. Previous work has added *single-core* RISC-V support to gem5 [13], and our work has focused on adding *multi-core* RISC-V support to gem5.

In Section 2, we describe our modifications to gem5 to support simulating multi-core RISC-V systems. Sections 3 and 4 present our functional and timing validation of the implementation. In Section 5, we describe the applications used to evaluate our work. Section 6 shows the performance of gem5. Section 7 presents a small design-space exploration study on a heterogeneous multi-core system with two different task-parallel programming frameworks using the RISC-V implementation in gem5.

2 ADDING MULTI-CORE RISC-V SUPPORT TO GEM5

In this section, we describe our modifications to gem5 to support the thread-related system calls (e.g., `clone`, `futex`, and `exit`) and RISC-V synchronization instructions (e.g., atomic memory operation, load-reserved, and store-conditional instructions) that are required to run multi-threaded applications in the simulator.

2.1 System Call Support

gem5 supports two modes of simulation: full-system (FS) and system-call-emulation (SE) [3]. In FS mode, applications execute using a simulated operating system (OS) exactly as they would on a real system. All system calls are trapped and handled by the simulated OS. In SE mode, system calls are directly emulated within the simulator itself. When an application executes a `write` system call, gem5 simply invokes a corresponding `write` system call using the host machine running the simulator, and no OS code is simulated. In this work, we focus only on SE mode. The implementation of system calls in SE mode is mostly ISA-independent, so much of this code can be directly reused to support RISC-V.

Each CPU in gem5 has a number of hardware (HW) threads. When an application executes, each software (SW) thread is mapped to a particular HW thread. A HW thread maintains its corresponding SW thread's state including its program counter (PC), its register values, and whether the SW thread is active. Thread creation, synchronization, and termination are handled through three system calls: `clone`, `futex`, and `exit`. So to run multi-threaded applications in SE mode, we must focus on supporting these three key system calls. Other thread-related system calls such as `gettid`, `getgid`, and `getpid` are completely ISA-independent and so are implemented in gem5 for RISC-V by default.

`clone` System Call – In Linux, an application spawns a new thread by calling the `clone` system call. The new thread can share resources with its parent thread, including the virtual memory space, file descriptors, and other process attributes. The sharing is specified through different flags (e.g., *CLONE_VM*, *CLONE_FILES*, and *CLONE_THREAD*) given to the system call. If the *CLONE_CHILD_CLEARTID* flag is set, then when a child SW thread terminates it should wake up its parent SW thread.

When executing the `clone` system call in gem5's SE mode, the simulator first finds an available HW thread. Pointers to shared resources (e.g., page table and file descriptor table) are copied from the calling SW thread to the new one. If non-shared resources (e.g., stack space and thread local storage) are pre-allocated, pointers to these resources will be passed into the `clone` system call and then used to initialize the SW thread. Otherwise, gem5 will allocate such resources on its own. After all necessary attributes and resources are initialized, gem5 activates the HW thread context, and the new SW thread starts executing its first instruction. Most of the existing implementation of the `clone` system call was leveraged to support RISC-V. We implemented some RISC-V specific requirements including a different system call API and register file initialization process.

`futex` System Call – Linux supports OS-level thread synchronization through the `futex` system call. The system call supports two operations: *FUTEX_WAIT* and *FUTEX_WAKEUP*. When a SW thread executes the *FUTEX_WAIT* operation, the SW thread checks if the value at a given address still matches a given expected value. If so, the SW thread waits by sleeping. A different SW thread can execute the *FUTEX_WAKEUP* operation to wake up one or more SW threads waiting on a given address. The *FUTEX_WAIT_BITSET* and *FUTEX_WAKE_BITSET* flags enable a SW thread to use a bit map to control which waiting thread(s) to wake up when performing the *FUTEX_WAKEUP* operation. The bit-set flags are commonly used in some parallel programming frameworks (e.g., OpenMP and Cilk).

In gem5's SE mode, each futex address is associated with a list of waiting HW threads. To execute the *FUTEX_WAIT* operation, gem5 puts the calling HW thread into a thread list associated with a given futex address and then suspends the HW thread. The suspended HW thread becomes idle. When a SW thread executes the *FUTEX_WAKEUP* operation on the same address, some HW threads waiting in the thread list are woken up and re-activated. The implementation of the `futex` system call is ISA-independent, so we only needed to modify it to support the bit-set flags to selectively wake up threads. We also needed to fix a more fundamental issue in the thread suspension and activation logic used by all gem5 CPU models. More details on our modifications are described in Section 3.

`exit` System Call – A SW thread calls the `exit` system call to terminate its execution. If the *CLONE_CHILD_CLEARTID* flag was used to `clone` the child SW thread, then the parent SW thread needs to be woken up.

In gem5's SE mode, when a SW thread executes the `exit` system call, gem5 cleans up all micro-architectural and architectural state belonging to the thread in the CPU pipeline. It then detaches the SW thread from its current HW thread, and the HW thread becomes available for future use. If waking up its parent thread is required, gem5 performs the *FUTEX_WAKEUP* operation on an address given to the `clone` system call that was used to create this SW thread.

2.2 Synchronization Instruction Support

The RISC-V "A" standard extension for atomic instructions supports two types of synchronization instructions: atomic memory operations (`AMO`) and load-reserved/store-conditionals (`LR/SC`) [11]. RISC-V supports the release consistency model and a memory fence instruction (`FENCE`). These instructions and the memory model are used to synchronize threads through shared variables. Although they have been recently implemented in gem5, their functionality was only validated for single-core simulations [13]. In multi-core simulations, we found that some executions using synchronization instructions implemented in the previous work could lead to race conditions and/or thread starvation. We describe our modifications to the implementation to fix these issues.

`AMO` Instructions – `AMO` instructions (e.g., `amoadd`) perform read-modify-write operations atomically to a given address. They appear to be executed in a single step with respect to all threads.

There are two ways to implement `AMO` instructions: (1) locking a target cache line before performing the operation using the CPU pipeline; and (2) embedding AMO arithmetic logic units (ALU) inside private L1 caches [12]. We chose the second approach to implement `AMO` instructions in gem5. We modified its cache model to support executing ALU operations directly in caches. We added a new memory request type called *atomic* in addition to *load* and *store*. *Atomic* requests are treated as if they were normal *store* requests except that no data-forwarding between an *atomic* request and a subsequent *load* request to the same address is allowed. This is due to the fact that *atomic* requests carry no valid data until they are executed in caches. Similar to *store* memory requests, an *atomic* request requires exclusive access to its target cache line through the cache coherence protocol before updating the line in an L1 cache. The cache then executes the request's ALU operation and updates the cache line in one step. The exclusive access and one-step execution inside the cache guarantees the atomicity of the `AMO` instruction. The previous value at the target address is returned to the executing CPU pipeline after the *atomic* memory request is completed in the cache.

`LR/SC` Instructions – An `LR` instruction reserves exclusive access to a shared address. An `SC` instruction performs an update to the value at the shared address only if there is still a valid reservation on the address. The pair of instructions is commonly used to perform lock-free atomic read-modify-write operations. Using an `LR/SC` instruction pair to synchronize multiple threads is prone to

aq	*rl*	Ordering Semantics	Micro-op Sequence
0	0	Relaxed	AMO/LR/SC
0	1	Releasing	fence; AMO/LR/SC
1	0	Acquiring	AMO/LR/SC; fence
1	1	Sequentially consistent	fence; AMO/LR/SC; fence

Table 2: Micro-Operation Sequences for AMO and LR/SC Instructions – Each sequence corresponds to a configuration of *aq* and *rl* bits set in the instructions and a memory ordering rule in the release consistency model.

livelock. An SC instruction executed by thread A may never succeed if an LR instruction executed by another thread continually invalidates thread A's reservation. RISC-V guarantees an SC instruction will eventually succeed under certain constraints on the number and type of instructions between an LR/SC pair [11].

The implementation of LR/SC in gem5 maintains a per-HW-thread list of reserved addresses. When an LR instruction is executed in a HW thread, a snoop request is placed on a cache coherence bus to revoke any reservation of the instruction's target address. Once all reservations in other HW threads are invalidated, the address is pushed into the requesting thread's reservation list. Later, when executing an SC instruction, the HW thread checks if the instruction's target address still exists in the thread's reservation list. If so, the SC instruction succeeds, and the address is popped off the list. Otherwise, the instruction fails. If a HW thread receives a snoop request for an address, it revokes any matched entry in its own list. We made the reservation list structure private for each HW thread to correctly support LR/SC in multi-core simulations. To implement RISC-V's livelock freedom guarantee, we modified the L1 cache to hold off processing LR snoop requests to an address for a bounded period of time if there is an active reservation on the address.

Release Consistency Model – RISC-V supports a release consistency model [6]. Under the model, memory operations are free to be re-ordered unless there is a memory fence (FENCE) instruction between them. RISC-V also supports two memory ordering flags (*acquire*, *release*) encoded in two corresponding bits (*aq*, *rl*) inside AMO and LR/SC instructions. The *acquire* flag prevents memory operations *after* an AMO or LR/SC instruction from being re-ordered with respect to the instruction. The *release* flag prevents memory operations *before* an AMO or LR/SC instruction from being re-ordered with respect to the instruction.

The RISC-V FENCE instruction is implemented in the current version of gem5. Its implementation prevents memory instructions after the FENCE instruction from being issued until all memory instructions before the FENCE instruction retire. To implement the memory ordering bits embedded in AMO and LR/SC instructions, we used gem5's micro-operation feature that allows breaking an instruction into a sequence of smaller micro-operations to be executed by the CPU pipeline. Depending how *aq* and *rl* are set in an AMO or LR/SC instruction, we inserted a fence micro-operation(s) before and/or after the AMO or LR/SC instruction. Table 2 shows all four configurations of *aq* and *rl* bits, their memory ordering semantics, and their corresponding sequences of micro-operations.

3 FUNCTIONAL VALIDATION

In this section, we describe our functional validation of the RISC-V implementation in gem5. We first show a major challenge with using gem5's current regression tests to validate the implementation. We then explain our approach and describe how we applied it to validate the functionality of thread-related system calls and RISC-V instructions.

Challenge – Although gem5 already has a regression test suite including some C/C++ benchmarks and their reference outputs, using these tests to debug a complex CPU model is challenging. A C/C++ benchmark, even a very simple one, can compile to thousands of instructions. Since a compiler can optimize the benchmark, the generated assembly code is often hard to understand. When the benchmark fails, tracing the problem through the large number of instructions is difficult and time-consuming. Debugging a multi-core CPU model that runs multi-threaded applications is even worse. A problem can appear to happen in a code region that is far from where the actual bug occurs. Therefore, we need a better approach to validate functionality in gem5.

Approach – Instead of using C/C++ benchmarks to validate a model in gem5, we used extensive, well-crafted assembly and low-level C unit tests. Each small test written in assembly code stresses a single instruction or system call without extra complexities coming from any C/C++ library and compiler. We used low-level C unit tests to discover missing functionality that is used in real libraries (e.g., GNU pthread library). By thoroughly testing an implementation at a low level, we can be more certain about the correctness of each instruction and system call.

Implementation – We applied the approach to validate functionality of the single-threaded and multi-threaded implementation of RISC-V in gem5.

For the single-threaded implementation, we leveraged an extensive assembly test suite in the open-source RISC-V tool chain[1]. The RISC-V test suite is designed to run on bare metal systems without any OS support, and it communicates to a host machine to inform test outputs. However, gem5 simulates systems with OS support, so to integrate the suite into gem5, we added a new testing environment that ignores the initial to-host communication setup in the original suite and calls the exit system call with an exit status number denoting which test case fails.

For the multi-threaded implementation, we built our own assembly and low-level C unit tests. Since we do not want to have the complexity of threading libraries (e.g., GNU pthread library) in our assembly tests, we wrote a minimal threading library written in assembly code to simplify developing new multi-threaded assembly tests. The library includes minimal functionality to create, synchronize, and terminate threads using the clone, futex, and exit system calls. We first validated the implementation of these system calls. Then we built new tests using the minimal library to validate the implementation of AMO and LR/SC instructions on a multi-core system. The multi-threaded tests are focused on inducing potential race conditions and other synchronization bugs that are impossible to detect in single-threaded tests. Low-level C unit tests were built to detect missing functionality used in the GNU pthread

[1]https://github.com/riscv/riscv-tests

library. Each unit test is focused on a single pthread function (e.g., `pthread_create`, `pthread_join`, and `pthread_mutex_lock`).

Using our approach, we were able to detect and fix numerous bugs in gem5's CPU models efficiently. Some of the bugs are related to incorrect suspension and resumption of HW threads in a CPU pipeline, which would be hard to reveal, trace, and fix using only C/C++ benchmarks. Some other bugs happened in the out-of-order CPU pipeline's memory disambiguation unit and load/store queue. Without the ability to control interactions between memory instructions, it would be challenging to reproduce and trace such memory-related bugs. There were a couple of bugs related to incorrect interpretation of the `clone` system call's API. They were easily detected in our simple assembly tests.

4 TIMING VALIDATION

Detailed CPU models in gem5 are meant to be used as generic models and are not validated against an actual cycle-accurate micro-architecture [9]. Users of the models often need to re-configure them and validate their performance against a target micro-architecture [7]. In this section, we first explain the challenges involved with timing validation for the RISC-V implementation in gem5, before describing a general approach for such timing validation. We then show an example of how we validated a multiplier unit in gem5's in-order CPU model against the multiplier unit in the Rocket chip.

Challenge – Timing or performance validation of a CPU model in gem5 is often performed using C/C++ benchmarks (e.g., SPEC CPU2006) [7]. Performance counters (e.g., the total number of cycles and instructions) are used to compare the performance of the simulated system vs. the target system. There are two main problems with this approach. First, it is often challenging to detect a performance bug. Since this approach relies on very general performance statistics of high-level benchmarks, different simulation errors and performance bugs can together skew the overall performance results [9]. Second, parameters of a model can be tuned to make the model appear to have correct timing behavior only in a small set of benchmarks [9]. When running the model with a benchmark that heavily uses HW units that are not validated, the model's performance may become incorrect.

Approach – Instead of using C/C++ benchmarks to validate the performance of a whole CPU model in gem5, we argue for an incremental validation approach using assembly micro-benchmarks. Each micro-benchmark is carefully designed to validate a specific HW unit (e.g., branch predictor, multiplier, decoder, and memory load/store queue) using a sequence of instructions that heavily use the target unit. The sequence's performance is measured through HW cycle and instruction counters. Some techniques including cache warm-up and loop unrolling can be applied to minimize interference from other HW units.

Implementation – In this work, we applied the approach to validate the multiplier's performance in gem5's in-order CPU model against the multiplier in a Rocket chip. We configured the Rocket chip generator to generate an in-order CPU model that has an 8-cycle iterative multiplier. We wrote a micro-benchmark that executes 500 `mul` instructions back-to-back with minimal read-after-write dependencies. We did not use branch instructions to loop through the sequence to prevent the branch predictor from affecting the sequence's timing behavior. We also warmed up the instruction cache by pre-executing the sequence to minimize interference from the memory system. We used `rdcycle` and `rdinstret` instructions to count the number of cycles and dynamic instructions in the sequence of interest.

Metrics	gem5's In-Order Model	Rocket In-Order Model
DInst	503	503
CPU Cycle	5010	5003
CPI	9.96	9.95

Table 3: Timing Validation of the Multiplier Unit in gem5's In-Order CPU Model against the Multiplier in the Rocket Chip – Performance numbers are for the sequence of 500 `mul` instructions. DInst = Dynamic Instruction. CPI = Cycles Per Instruction.

For each `mul` instruction, the Rocket core's iterative multiplier spends one cycle taking input values in, eight cycles doing the multiplication, and another cycle pushing the output value to the next pipeline stage. In total, each `mul` instruction takes 10 cycles to complete in the Rocket core's multiplier. The multiplier is not pipelined, so it cannot execute new `mul` instructions until the current one completes. To model this iterative multiplier in gem5, we configured gem5's in-order CPU's multiplier unit to have 10-cycle issue and execution latency. Table 3 shows performance numbers for both models after the validation. The multiplier in gem5's in-order model performed close to the multiplier in the Rocket core in terms of the instruction throughput.

This validation of the multiplier unit is a starting point, and future validation of other HW units in gem5's CPU models are necessary. Our work suggests an incremental timing validation approach that can be applied to gem5's CPU models.

5 EVALUATION WORKLOAD

We chose 13 applications from the Ligra benchmark suite [15] as our workload (see Table 4). Ligra is a graph processing framework designed for shared-memory systems. It supports multiple threading libraries including Cilk [8] and OpenMP [10]. Ligra provides two lightweight routines called *mapEdge* and *mapVertex* to process subsets of edges and vertices respectively. Multiple subsets can be processed in parallel. Many applications in Ligra show irregular characteristics in their parallelism and memory access patterns.

In terms of input graphs, we picked two real-world graphs called *socfb-American75*, which is from a collection of Facebook networks [14], and *Kneser_10_4_1*, which is from a sparse matrix collection [5]. *socfb-American75* is a dense graph which has around 6,000 vertices and 440,000 edges. *Kneser_10_4_1* is a sparser graph with around 350,000 vertices and 990,000 edges.

In this work, we used four different versions of each Ligra application: Serial, OpenMP-Static (OMP-S), OpenMP-Guided (OMP-G), and Cilk-Work-Stealing (Cilk-WS). We used the OpenMP and Cilk task parallel runtime libraries shown in Table 5. We used the open-source RISC-V tool chain including the GNU compiler and GNU libc to compile the Ligra applications. We only made a minor modification to the compiler to be able to compile the Cilk runtime. For the OpenMP runtime, the tool chain works out-of-the-box.

Applications	Input Graphs	DInst (M)			
		Serial	OMP-S	OMP-G	Cilk-WS
BC	kneser	1152	1148	1149	1196
BFS	kneser	502	502	502	523
BFSCC	kneser	607	988	1079	2595
BFS-Bitvector	kneser	1042	1059	1059	1082
Components	kneser	1719	1719	1735	1746
MIS	kneser	646	810	844	835
KCore	socfb	628	641	648	1009
PageRank	socfb	2858	2861	2862	3241
PageRankDelta	socfb	400	401	401	441
Radii	socfb	268	268	268	283
Triangle	socfb	1069	1069	1069	1121
BellmanFord	socfb	138	137	137	147
CF	socfb	2670	2670	2670	2889

Table 4: List of Ligra Applications – Serial = single-threaded versions. OMP-S = multi-threaded versions using OpenMP runtime with the static scheduling policy. OMP-G = multi-threaded versions using OpenMP runtime with the guided scheduling policy. Cilk-WS = multi-threaded versions using Cilk runtime with the work-stealing policy. DInst = dynamic instruction count. Multi-threaded versions are simulated on a 4-core in-order CPU in gem5. The reported numbers are only for regions of interest that include only graph computation phases and exclude input graph initialization phases in each application.

Runtime	Chunk Size	Task Assignment	Work Stealing
OMP-S	Fixed	Static	No
OMP-G	Adaptive	Dynamic	No
Cilk-WS	Fixed	Dynamic	Yes

Table 5: Threading Libraries Used in Our Experiments – Fixed chunk size = the task chunk size is fixed for a particular parallel region. Adaptive chunk size = task chunk size is adjusted dynamically to better handle workload imbalance within a parallel region. Static task assignment = tasks are assigned statically before entering a parallel region. Dynamic task assignment = tasks are assigned on the fly during the execution of a parallel region. If work stealing is available, a thread can steal tasks from other threads.

Metrics	gem5 Simulator	Chisel C++ RTL Simulator
DInst (M)	1125	1161
CPU Cycle (M)	1440	1956
KCPS	225	7
KIPS	175	4

Table 6: Performance Comparison between gem5 and Chisel C++ RTL Simulator – Both simulators run the same single-threaded binary of *KCore*. DInst = dynamic instruction. KCPS = kilo CPU cycles per second. KIPS = kilo instructions per second.

6 SIMULATOR PERFORMANCE

In this section, we show the performance of gem5 simulating four Ligra applications: *PageRank*, *PageRankDelta*, *KCore*, and *Triangle*. We first show a performance comparison between gem5 and the Chisel C++ RTL simulator. Then we show that using the fast-forwarding feature provided with gem5 increased its simulation speed up to 2× in the set of studied benchmarks. Finally, we present the scalability of gem5's performance in multi-core simulations. We ran our experiments on a multi-core machine with Intel Xeon E5620 CPUs running at 2.40 GHz.

6.1 gem5 vs. Chisel C++ RTL Simulator

We chose the Rocket chip generator as our baseline [1]. The generator is written in Chisel and can be configured to generate an RTL model in Verilog. Verilator is then used to compile the RTL model into a C++ cycle-accurate model that is significantly faster than the Verilog RTL model. We used a RISC-V proxy kernel to handle system calls executed in the Chisel C++ RTL simulator instead of simulating a full Linux kernel for better simulation performance. Unfortunately, the proxy kernel does not support the `clone` and `futex` system calls. Therefore, we are unable to perform a multi-threaded simulation performance comparison between gem5 and Chisel C++ RTL simulator. For this comparison, we used the gem5 configuration with the validated multiplier and the Rocket core as described in Section 4.

We chose the single-threaded version of *KCore* from the Ligra benchmark suite. We measured the end-to-end simulation time of both simulators using the `time` Linux command. We counted the number of cycles and instructions simulated in the simulators using `rdcycle` and `rdinstret` instructions. Table 6 shows a performance comparison between the two simulators. The Chisel C++ RTL simulator simulated slightly more instructions than gem5 due to the extra instructions required for executing system calls in the RISC-V proxy kernel. The Chisel C++ RTL simulator simulated roughly 36% more cycles than gem5 did. One possible reason for the difference is that despite using the validated multiplier in gem5, the performance of other HW units (e.g., branch predictor, memory load/store unit, and memory system) has not been validated.

Despite differences in the absolute instruction and CPU cycle counts, the average number of cycles per second and instructions per second provide intuition into the relative performance of both simulators. gem5 is more than an order of magnitude faster compared to the Chisel C++ RTL simulator. This large difference is one of the key benefits of cycle-approximate vs. cycle-accurate simulation. To provide context, simulating 1B instructions would take almost three days when using the Chisel C++ RTL simulator, but this same simulation would take less than two hours when using gem5.

6.2 Fast-Forwarding Simulation

gem5 can fast forward a sequence of instructions by simulating them with a simple CPU model that only captures functional behavior and excludes the timing behavior of the CPU pipeline, the memory system, or both. Table 7 shows two simple and two detailed CPU models that are available in gem5. Switching between a simple and a detailed model can happen on any given simulation tick. We modified gem5 to support a custom control-status register (CSR) to enable software running on the simulator to indicate when to switch into or out of a detailed CPU model. We used this CSR to fast forward our benchmarks during their input graph initialization phase.

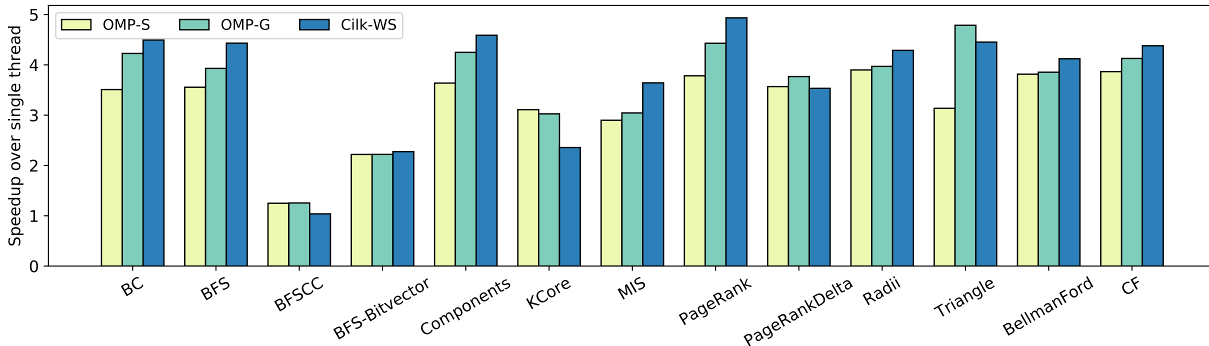

Figure 1: Performance of Different Task Scheduling Mechanisms in a Heterogeneous System

CPU Models	CPU Pipeline	Memory
AtomicSimpleCPU	Simple	Simple
TimingSimpleCPU	Simple	Detailed
MinorCPU	Detailed In-order	Detailed
DerivO3CPU	Detailed Out-of-order	Detailed

Table 7: Available CPU Models in gem5. – Simple models only simulates functional behaviors while detailed models capture both functional and timing behaviors.

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.

Table 8 shows performance improvements of gem5 when using fast-forwarding. All applications studied in this section used the same input graph, so they all had the same number of fast-forwarded instructions. Depending on the length of the initialization phase with respect to the full execution time, fast-forwarding results in a speedup of 1.16–2.01×.

6.3 Performance Scalability

To understand the performance scalability of gem5 in multi-core simulations, we ran the four benchmarks using the OpenMP runtime and the static scheduling policy on systems with a different number of in-order CPU cores. Figure 2 shows the average number of simulated instructions per second. The result shows that gem5's performance scales well with the number of simulated CPU cores. When simulating more CPU cores, gem5's does slows down a little bit since it simulates more thread communication events between cores.

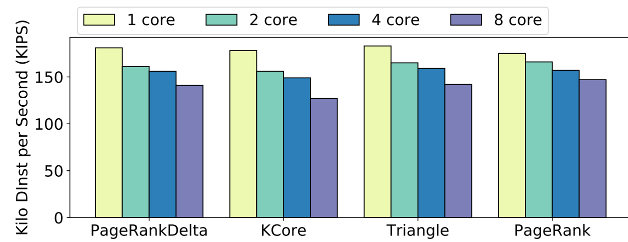

Figure 2: Performance of gem5 in Multi-Core Simulations

7 DESIGN SPACE EXPLORATION

In this section, we are interested in using the implementation of RISC-V in gem5 to study how irregular applications using different threading libraries and task scheduling policies perform on a heterogeneous multicore system. In particular, we are interested in the relative performance of static, guided, and work-stealing task scheduling policies for graph applications executing on a system with both simple and complex cores [16].

We used gem5 to model a quad-core cache-coherent RISC-V system with two simple in-order and two complex out-of-order cores. Each core has its own private L1 cache. Constructing this model is straight-forward in gem5 due to its modular design and simple Python-based configuration interface. We only needed to make minor changes in the Python configuration. In contrast, building such a system in RTL would be very challenging.

We ran all 13 Ligra applications with different task scheduling policies. Figure 1 shows the speedup of each configuration over the single-threaded version of Ligra applications. The multi-threaded versions of most applications except *BFSCC* performed significantly better than their single-threaded versions. In *BFSCC*, our selected input graph results in a large serial region. On average, the quad-core heterogeneous system achieved a 3.53× speedup over the single-core system.

Dynamic task scheduling policies (i.e., OMP-G and Cilk-WS) generally performed better than the static task scheduling policy. This is due to the workload imbalance in many graph applications and the heterogeneity of the studied system. Complex and simple cores complete tasks at different rates. A dynamic task scheduling mechanism helps balance the workload between cores, which helps

increase the overall throughput. In terms of performance, OMP-S is never the best choice for Ligra applications except *KCore* and *BFSCC*. Most of parallel regions in *KCore* are highly regular, and its tasks are light-weight. *BFSCC* has little parallelism, so its multi-threaded versions did not perform much better than its single-threaded version.

8 CONCLUSIONS

We presented our work on simulating multi-threaded RISC-V systems in gem5. We contributed an implementation of thread-related system calls and synchronization instructions to the existing RISC-V implementation in gem5. We also modified gem5's CPU models to simulate multi-threaded workloads correctly. We showed our validation approach and how we applied the approach to validate the functional and timing behavior of the implementation. We presented gem5's simulation performance in comparison to the Chisel C++ RTL simulator, the simulation speedup achieved by using the fast-forwarding feature in gem5, and gem5's scalability in multi-core simulations. Our implementation in gem5 can run real-world applications and task parallel runtime libraries including OpenMP and Cilk. We showed a small design space exploration to illustrate how gem5 can help system designers explore different design options quickly.

ACKNOWLEDGMENTS

This work was supported in part by NSF CRI Award #1512937, NSF SHF Award #1527065, NSF E2CDA Award #1740286, AFOSR YIP Award #FA9550-15-1-0194, SRC nCORE Task 2758.004, and donations from Intel. The authors acknowledge and thank Alec Roelke and Mircea R. Stan for their work on the initial single-threaded RISC-V gem5 port as well as their feedback on our own work. The authors also acknowledge and thank Christopher Torng and Shreesha Srinath for their advice on adding support for a new instruction set in gem5, Moyang Wang for his insight on porting software runtimes to RISC-V, and Khalid Al-Hawaj for his help in configuring the Rocket chip generator.

REFERENCES

[1] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman. The Rocket Chip Generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[2] K. Asanovic and D. Patterson. Instruction Sets Should Be Free: The Case for RISC-V. Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley, Aug 2014.

[3] N. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 Simulator. *SIGARCH Computer Architecture News (CAN)*, 39(2):1–7, Aug 2011.

[4] S. Davidson, S. Xie, C. Torng, K. Al-Hawaj, A. Rovinski, T. Ajayi, L. Vega, C. Zhao, R. Zhao, S. Dai, A. Amarnath, B. Veluri, P. Gao, A. Rao, G. Liu, R. K. Gupta, Z. Zhang, R. G. Dreslinski, C. Batten, and M. B. Taylor. The Celerity Open-Source 511-Core RISC-V Tiered Accelerator Fabric: Fast Architectures and Design Methodologies for Fast Chips. *IEEE Micro*, 38(2):30–41, Mar/Apr 2018.

[5] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, Dec 2011.

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. *Int'l Symp. on Computer Architecture (ISCA)*, pages 15–26, 1990.

[7] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, N. Paver, and N. K. Jha. Sources of Error in Full-System Simulation. *Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Mar 2014.

[8] Intel Cilk Plus Language Extension Specification, Version 1.2. Intel Reference Manual, Sep 2013.

[9] T. Nowatzki, J. Menon, C. H. Ho, and K. Sankaralingam. Architectural Simulators Considered Harmful. *IEEE Micro*, 35(6):4–12, Nov 2015.

[10] OpenMP Application Program Interface. OpenMP Architecture Review Board, 2008.

[11] RISC-V Foundation. `http://www.riscv.org`, 2018 (accessed Mar 17, 2018).

[12] Rocket Core Overview. `http://www.lowrisc.org/docs/tagged-memory-v0.1/rocket-core`, 2018 (accessed Mar 17, 2018).

[13] A. Roelke and M. R. Stan. RISC5: Implementing the RISC-V ISA in gem5. *Workshop on Computer Architecture Research with RISC-V (CARRV)*, Oct 2017.

[14] R. A. Rossi and N. K. Ahmed. An Interactive Data Repository with Visual Analytics. *SIGKDD Explor.*, 17(2):37–41, 2016.

[15] J. Shun and G. E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *Symp. on Principles and practice of Parallel Programming (PPoPP)*, Feb 2013.

[16] C. Torng, M. Wang, and C. Batten. Asymmetry-Aware Work-Stealing Runtimes. *Int'l Symp. on Computer Architecture (ISCA)*, Jun 2016.