A Universal Parallel Front-End for Execution Driven Microarchitecture Simulation

Chad D. Kersey
Georgia Institute of Technology
Atlanta, GA 30332
cdkersey@gatech.edu

Arun Rodrigues
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185
afrodri@sandia.gov

Sudhakar Yalamanchili
Georgia Institute of Technology
Atlanta, GA 30332
sudha@ece.gatech.edu

ABSTRACT

Execution driven microarchitecture simulators tend to devote a large portion of their source code to a front-end that performs instruction set level functional simulation, providing the decoded instruction stream to a back-end that performs timing simulation. In this paper we introduce the current incarnation of QSim, a universal front-end for execution driven multicore microarchitecture simulators. QSim adapts the popular and portable QEMU full-system emulator to a thread safe, instruction set neutral API, running unmodified application binaries in a lightly modified Linux operating system. QSim has been shown to support at least 512 emulated hardware threads, each running in a separate host thread.

Categories and Subject Descriptors

B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms

Performance, Measurement

Keywords

simulation, simulator front-ends, emulators

1. INTRODUCTION

New ideas in computer architecture are universally explored using simulators before they are implemented in hardware. The high cost and long turn-around time of building complete prototypes to test new ideas in architecture prohibits full-scale prototyping of incremental improvements. The benefits of quick feedback and manageable cost make simulators a fundamental part of the computer architect's toolbox. Further, processor design has increasingly relied on making thread-level parallelism available to the programmer. Recent CPU designs [18, 17] rely on multithreaded, multicore designs to expose TLP, and simulators have to be able to both simulate these kinds of processors and perform well when run on these kinds of processors.

Modern simulators are typically designed in two parts; a *front-end* provides a high-fidelity implementation of the instruction set, ensuring correct execution of the *guest* application, and a *back-end* provides timing models and records data. The front-end is typically a large portion of the source code of microarchitectural simulators, but represents work that often does not have to be replicated for each new simulator. The functional simulation component in execution driven simulations is typically an emulator like Simics [12, 10] or QEMU [4, 20], but can also be a native executable instrumented using a tool like Pin [11, 13].

Of the features provided by these current solutions for building simulation front-ends, it was decided that those most important to our productivity were:

- Supporting the creation of both execution-driven and trace-driven simulators.
- Allowing the construction of multithreaded simulators simulating multithreaded CPUs.
- Providing an API that simplifies both writing new back-ends and modifying the front-end.
- Allowing the simulation of both operating system and application code.
- Exposing instruction-level information and control of execution to the back-end.

These traits may be available to varying degrees in the one-off front-ends developed for the simulators listed above, but to the authors' knowledge, none has provided all of them in a single package. It is the unique combination of these traits that is the contribution of QSim.

The QSim project, introduced in [16], provides such a front-end, running at about 20MIPS. The QSim API is designed to be easy to connect to a diverse range of back-ends and be used by other front-ends as well. This has already been demonstrated with a trace reader that exports the same API as the QSim emulator. Back-ends developed using the QSim API so far include simulators, trace gathering programs, and an interactive debugger. Just as compiler infrastructures have managed complexity by lowering programs through a series of intermediate representations to code generation, the QSim API is intended to be a similar step toward the simplification of the construction of full system many core simulators by emphasizing the interface

RAPIDO '12, January 23 2012, Paris, France

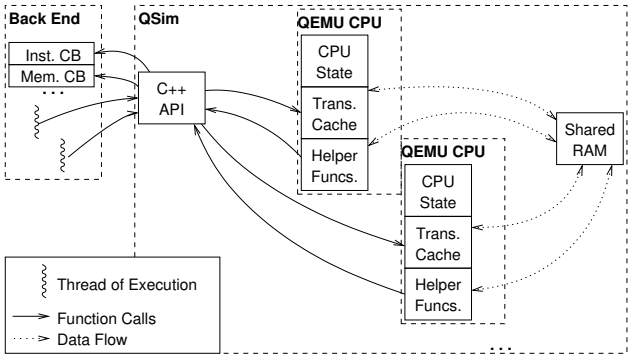

Figure 1: Diagram of QSim software architecture. Multithreaded back-ends can make parallel calls to the API which result in parallel calls to the QEMU CPU and potentially parallel calls to callbacks in the back-end.

between simulator front-ends and back-ends. This reduces the work required to write a simulator and creates a nucleus around which a multitude of interchangeable front-ends and back-ends can be created.

2. QEMU

QEMU [4] is a robust emulation infrastructure using dynamic binary translation, an effective technique for fast functional simulation. Dynamic binary translators perform just-in-time compilation of the guest CPU's instructions into emulator code to be run on the host. A *translation cache* stores already-translated segments of guest code so that repeatedly executing the same blocks does not require repeated translation of those blocks, taking advantage of temporal locality to improve emulation performance.

QEMU implements this technique in two stages, translating the guest instructions first to a set of operations in a low-level intermediate format (LIR) and using the Tiny Code Generator (TCG) to translate this LIR into host instructions. This simplifies the act of porting QEMU to new host and guest instruction sets. QEMU is therefore portable by design, already supporting a variety of guest instruction sets including x86, ARM, and MIPS.

Because of the level of fidelity of the QEMU CPU emulator and its open source license, it is an inviting choice for use as a simulator front-end. However, since its intended use is as an emulator and virtual machine monitor, QEMU's design is not readily usable as a simulation front-end. The few simulation projects that have used QEMU to drive simulators have done so with considerable effort; effort which is duplicated by each new project. QSim adapts QEMU for simulation in a generic way, providing a rich API which allows control of the QEMU CPU emulator at an instruction level and parallelization at the CPU level. In doing so, the current version of QSim abandoned QEMU's support for I/O devices, the built-in kernel loader, and checkpointing, replacing these with simpler, but less general functionality.

3. QSIM

QSim is implemented as library with a C++ API that manages a collection of instances of a modified QEMU CPU emulator. Each instance contains its own set of global variables, including the translation cache and CPU state, but shares a common host process, guest RAM state, and QSim callback pointers. This loosely-coupled implementation is completely thread safe, and the QEMU CPUs can be run either simultaneously from different host threads, or in turn from a single thread. Figure 1 is a diagram of the QSim implementation.

3.1 Implementation

QSim's instances of QEMU have been modified to place calls to new, QSim-specific "helper" functions in the translation cache along with the translated guest code. From these helper functions, user-configured callbacks are called as required. The set of events in execution that can cause a callback to be called are enumerated in Table 1.

QSim provides a complete simulation environment with guest utilities, a port of Linux, and ready-to-run benchmarks from a variety of suites. QSim is capable of running unmodified binaries on a lightly modified (392 modified lines) Linux kernel. Because QSim instantiates the QEMU CPU emulator multiple times within the same process, the QEMU implementations of PC IO devices have been abandoned, limiting the devices emulated by QSim to a set of processors connected to a shared memory. This eliminates the need to synchronize the large set of emulated hardware otherwise provided by QEMU when running QSim's emulated CPUs in parallel.

Because all of QSim's instruction set emulation logic is implemented in QEMU, which is designed with portability in mind [4], QSim itself is portable to new architectures. Currently, multi-core guests are only supported on x86-32, but x86-64 single-core guests have been booted, and multi-core x86-64 guests require only further modifications to Linux in order to work. There is also a project which aims to port QSim to ARM guests, allowing the same set of back-ends to work with an entirely new instruction set.

3.2 Thread Safety and Parallelization

The QSim library is thread safe. Different guest CPUs can be run simultaneously from different host threads with no concern for the correctness of execution. The guaranteed atomicity of certain memory operations is preserved by the implementation, even though the QEMU emulators themselves do not preserve atomicity. This allows QSim to be used in parallel simulators, perhaps using the technique of conservative parallel discrete event simulation [8]. It has also been used to gather traces in parallel, by writing a trace from each emulated CPU to a separate file.

Since the back-ends of microarchitectural simulators typically use more host CPU time than the front-end, enabling parallelism through thread safety in the API is more important than having a parallel front-end. MARSS, a fast detailed microarchitecture simulator with a QEMU front-end, runs at about 200kIPS [14], while QSim runs at around 20MIPS per thread, implying that emulation contributes at most one percent to the execution time of MARSS or similar simulators. QSim is, however, a parallel front-end. Other back-ends may be able to take advantage of additional performance in the front-end, and parallelism in the guest software is exploited by QSim.

3.3 Saving and Restoring State

QSim provides the ability to start from a saved state file instead of starting at boot time. This can be important for simulation, because large simulated core counts lead to increasingly large boot times. QSim has been demonstrated to boot hundreds of cores, but getting through the Linux boot process for this many cores can take hours. On our test machine, booting 128 cores required almost two hours. For this reason, QSim provides a set of already-booted kernel images for core counts from 1 to 512 in power of two increments that can load guest programs through a block transfer interface from the host file system. This allows simulators to bypass the boot process when necessary.

The saving and loading of state is handled entirely through the QSim API. The underlying QEMU emulator provides similar functionality, which was reimplemented at the QSim level to reduce the number of necessary modifications to QEMU.

4. AN API FOR FRONT-ENDS

As a central part of QSim we have developed an instruction set agnostic API for interfacing functional simulator front-ends to back-ends, including trace gathering programs, execution analysis and visualization utilities, and microarchitectural simulations. The QSim API provides a pay-as-you-go performance penalty for instrumenting guest code. Instrumentation can be as simple as a callback received for each instruction executed on the guest and as complicated as examining memory and register contents every time a read or write occurs, implementing new behavior by adding instructions to the guest instruction set, and simulating I/O devices complete with interrupts. The incremental cost of various instrumentation can be seen for a simple trace gatherer in Section 5.2.

The target back-end for QSim is the microarchitecture simulator. A microarchitecture simulator using QSim would typically:

1. Instantiate the front-end.
2. Set up callbacks.
3. Advance guest CPUs as needed (e.g. call `run()`).
4. Use information gathered from callbacks to make timing decisions.
5. If the simulation has not finished, repeat from step 3.

A simple simulator might, for example, execute a number of instructions in each guest CPU, store information gathered in the callbacks in a buffer, and use that buffer's contents to compute the state of the pipeline for the following cycles.

4.1 Instantiating the Front-End

A front-end implementing the QSim API is encapsulated within a singleton object called `OSDomain`. An `OSDomain` includes a set of CPUs and provides functions to control them, including reading and writing memory locations and setting callbacks. The object is called an `OSDomain` because it is presumed that all of the CPUs within it share memory and thus an operating system image.

Arguments to the constructor for `OSDomain` are perhaps the single most variable aspect of front-ends that implement the QSim API. For trace readers, a trace file or set of trace files would be given to this. In QSim, the constructor takes either a number of emulated CPUs and a kernel image, as shown in line 1 of Figure 2 or a file containing a saved machine state.

```
1  OSDomain osd(4, "bzImage");

3  osd.set_inst_callback(obj, &Class::iCB);
   osd.set_mem_callback(obj, &Class::mCB);
5
   while (cond) {
7    for (int j = 0; j < 1000; ++j) {
       for (int i = 0; i < osd.get_n(); ++i)
9        osd.run(i, 1000);
     }
11     osd.timer_interrupt();
   }
```

Figure 2: Interacting with the QSim API.

4.2 Callbacks

The basic way the QSim front-end communicates to its back-end is through a set of callbacks. These are in turn called by "helpers;" functions which can be called from with the QEMU translation cache. The major modifications to QEMU required by QSim are a set of helper functions used to call callbacks and modifications to the QEMU translation process to support them. As shown on lines 3 and 4 of Figure 2, these can be arbitrary member functions of any object, as long as they have the appropriate prototype. A complete list of callback types supported by QSim is enumerated Table 1.

4.3 Controlling Execution

Line 9 of Figure 2 demonstrates how the front-end is advanced. The `OSDomain::run()` function takes as arguments a CPU number and a number of instructions. While QSim is capable of running for a precise number of instructions, front-ends using the API may not be able to run for exactly the number of instructions given, depending on their implementation. The instruction callback can be used to keep an accurate count.

The example in Figure 2 runs 1000 instructions on each CPU in turn until they have all executed one million instructions, and then calls `OSDomain::timer_interrupt()`. While new instructions, explained in the next section, can be used to provide other interfaces for timers to the guest, the only required API function to communicate the advancement of time to the guest is the timer interrupt function, which should be called periodically so the guest OS can update timing structures and run the scheduler as needed.

4.4 Expanding the Guest Instruction Set

The magic instruction callback mentioned in Table 1 is a method borrowed from the COTSon [1] simulation environment to expand the instruction set of the guest machine and allow out-of-band communication between the guest and the simulator back-end running on the host. This can be used to implement new synchronization primitives, high-resolution timers, simulated hardware, and wide-bandwidth communication between the guest and host environment. For example, it is used by the block transfer interface mentioned in Section 3.3 to coordinate the transfer of files into the guest's

Callback	Arguments	Description
Instruction	CPU, Virt./phys. address, Inst. code, Inst. type	Received at start of every instruction executed.
Register Access	CPU, Reg num./flag vec., size (0 for flags), read-/write	Register access. Size is set to zero and reg num contains a bit vector for condition code accesses.
Memory Access	CPU, Virt./phys. address, size, read/write	Accesses to RAM other than instruction fetch.
Atomic	CPU	Received at start of instruction (after instruction callback) if all memory operations in instruction are to be considered atomic by the memory system.
Interrupt	CPU, vector	CPU is processing an interrupt starting at the next instruction.
Magic instruction	CPU, magic instruction number	Used to extend guest instruction set.

Table 1: Callback types available in QSim.

RAM filesystem from the host.

5. BACK-END EXAMPLES

The QSim API has enabled the creation of a diverse set of back-end programs that can all share the QSim front-end. These include a simple trace collector, an interactive debugger, visualization tools, and an experimental microarchitectural simulator. Because they were all created to use the QSim API, they could be ported to other front-ends without major changes to their source code. In fact, of the programs developed so far, only the trace collector and debugger would require any changes to be fully functional when ported to a different instruction set, because they use an x86-only disassembler, Distorm [7] to provide disassembly of guest instructions.

5.1 Microarchitecture Simulation

Figure 3 shows a set of callbacks that is used to split an instruction stream into a plausible sequence of micro-instructions that are executed by the experimental SimpleSim out-of-order processor simulator. While the set of micrp-ops (μops) used by SimpleSim do not correspond to any vendor-defined set, they provide a reasonable approximation for the purpose of simulation that is not tied to any particular instruction set. With a few exceptions, like instructions using the `repz` prefix on x86 which combine control flow, memory accesses, and arithmetic, a vast majority of instructions on currently popular CPUs can be broken down into a series of memory loads and stores and arithmetic instructions. These μops consist of an operation type, corresponding to one of the QSim-defined, architecture-specific instruction types, a set of input (source) registers, and set of output (destination) registers.

The instruction types are themselves changeable, corresponding to functional unit types in the simulator, allowing for the simulation of processors with various instruction sets, including floating-point and vector instructions as well as traditional RISC instruction sets and even simple CISC instruction sets like x86.

These callbacks are amenable to modification depending on the desired characteristics of the machine being modeled. For example, this example breaks all load or store instructions into two micro-ops. A simple modification would be

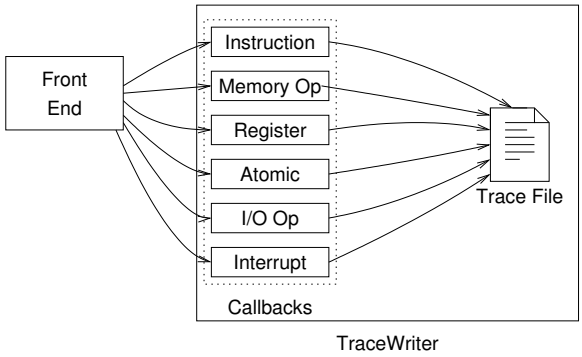

Figure 4: Trace collection using QSim is a simple matter of setting callbacks and formatting the output.

to check the number of members in the `inputs` and `outputs` sets at the beginning of `mem_rd()` and `mem_wr()`. If there has only been a single input register read (for `mem_rd()`) or two (for `mem_wr()`), then the operation can be considered as a simple load or store, and as such only require a single micro-op. An alternative used in the `utrace` micro-op trace generator is to consider single memory accesses to be part of the same micro-operation as the computation, adding a memory location as another source.

5.2 Trace Collection

The simplest application for a front-end simply sets every callback that QSim provides and dumps the information provided by them to a text file as shown in Figure 4. An implementation complete with disassembly of the dynamic instruction trace has been built with only 200 lines of code, most of them devoted to formatting the textual output as shown in Figure 5.

Trace collection in QSim with this simple program is slow, limited to around 400kIPS on our test machine, a Nehalem clocked at 2.6GHz, by the speed of the C++ formatted IO primitives, making the breakdown of trace time in Figure 6 mostly an illustration of the relative amounts of total output text devoted to each type of information. Traces gathered

```
inst(addr, length, type) {
    if (¬first_uop) {
        emit_uop(inputs, outputs, t);
    }

    t ← type;
}

reg_rd(reg) {
    inputs ← inputs ∪ {reg};
}

reg_wr(reg) {
    outputs ← outputs ∪ {reg};
}
```

```
mem_rd(addr) {
    outputs ← outputs ∪ {TMP_REG};
    emit_uop(inputs, outputs, MEM_RD);
    outputs ← ∅;
    inputs ← {TMP_REG};
}

mem_wr(addr) {
    outputs ← outputs ∪ {TMP_REG};
    emit_uop(inputs, outputs, t);
    inputs ← {TMP_REG};
    t ← MEM_WR;
}
```

Figure 3: A set of callbacks to split instructions into generic micro-ops.

```
2: Inst@(0x80482a9/0x7fd352a9, tid=0, USR[IDLE]):\
  RET  (QSIM_INST_RET)
2: Reg RD 4: 32 bits.
2: Mem RD(0xbfadfa20/0x7f81ba20): 32 bits.
2: Reg RD 4: 32 bits.
2: Reg WR 4: 32 bits.
2: Inst@(0x80482c2/0x7fd352c2, tid=0, USR[IDLE]):\
  JMP 0x14 (QSIM_INST_BR)
2: Inst@(0x80482d6/0x7fd352d6, tid=0, USR[IDLE]):\
  MOV EAX, 0x0 (QSIM_INST_NULL)
2: Reg WR 0: 32 bits.
```

Figure 5: Three instructions worth of trace output from the simple trace writer, including instruction disassembly, register accesses, and memory accesses.

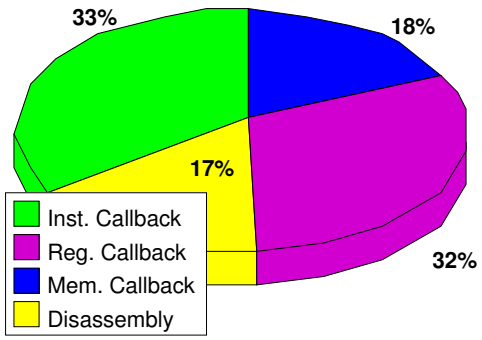

Figure 6: Execution time of text trace gatherer by activity. The emulator itself uses less than one percent of total execution time, as the program is limited by the speed of the C++ formatted I/O primitives.

in this way can be used to drive simulators, but in multi-threaded simulations represent only a single correct execution out of many possible ones, and one that would not have been likely to occur had the processors been timed with an appropriate simulator instead of all advancing an instruction at a time in lockstep. Traces of race-free programs annotated with the details of their synchronization operations do not necessarily suffer from this limitation [15], and using the magic instruction interface mentioned in Section 4.4, this kind of annotation could be added to traces gathered with QSim.

A binary trace format has also been created, along with a binary trace reader using the QSim API as mentioned in Section 1. This provides about 1MIPS per thread of trace reading or writing speed on our test machine. This is 20 times slower than the QSim emulator itself and ultimately limited by the speed of disk.

Compared to the popular Pin [11] binary instrumentation tool, there are some advantages to using QSim to gather traces.

- Physical addresses are provided for all instructions and loads/stores.
- Operating system code is emulated and traced as well.
- More details are exposed, including instruction types and register accesses.

There are several advantages to trace gathering with QSim over the alternative approach of using the Pin [11] binary instrumentation tool. While Pin allows gathering traces of application binaries, it does not allow tracing of operating systems.

5.3 Interactive Debugging

Debugging the early boot process or debugging software designed to interact with hardware that has not yet been implemented requires the use of emulators. The QSim debugger provides an interactive console for any emulator implementing the QSim API. Output from a QDB session is shown in Figure 7.

QDB has been an essential utility in modifying Linux to run as a QSim guest, since it allows interactive use of many of the emulator's features and manipulation of the guest

```
QDB: The QSIM Multiprocessor Debugger
=====================================

qdb> l System.map
Loaded symbols from file "System.map".
qdb> run 10
qdb> c
CPU 0:
 rax=00000fe0  rcx=00000000  rdx=000040c0  rbx=00000000
 rip=00010074  rsp=00001000  rbp=00000000  cr3=00000000
 TID=0 (idle)
 Nearest symbol: VDSO32_SYSENTER_RETURN
CPU 1:
 rax=00000000  rcx=00000000  rdx=00000623  rbx=00000000
 rip=00000000  rsp=00000000  rbp=00000000  cr3=00000000
 TID=-1 (idle)
 Nearest symbol: VDSO32_PRELINK
qdb> disas 0x10074 16
00010074: TEST BYTE [ESI], 0x11
00010077: ADD AL, [EAX+0x168b0474]
0001007d: AND AL, 0x2
0001007f: DB 0x81
00010080: RET 0x200
00010083: DB 0x73
qdb>
```

Figure 7: QSim Debugger session. A symbol map is loaded from a file, all running CPUs are advanced by ten instructions, register contents are examined, and then a part of guest RAM at the boot CPU's program counter is disassembled.

CPUs and register states. It also provides a simple way to profile guest code in terms of which functions, in both the OS and user programs, are executed for the most dynamic instructions. While this statistic can be misleading due to the variable nature of instruction execution times, few other options are available to quickly profile operating system code without modification.

5.4 Visualization

Figure 8 is the output of a program that plots memory accesses over program execution measured in dynamic instructions, both instruction and data, using the instruction and memory operation callbacks provided by QSim. These kinds of images make concepts such as locality and complexity visible, both within user code and the operating system. Figure 8 is a plot of memory accesses in a single-threaded program performing merge sort. The $O(n \log n)$ time complexity and $O(n)$ space complexity of the algorithm are readily apparent, as is the self-similarity caused by its recursivity.

6. RELATED WORK

The QSim front-end shares features with several other simulation projects. PTLSim [20] and the PTLSim derivative MARSS [14] both use an instrumented QEMU. Neither of these simulators exploit parallelism on the host, but both of them have an interface between front-end and back-end that could be mapped to a subset of the QSim interface. FAST [5], also built on QEMU, exploits parallelism on the host, speculative execution, and FPGA-based hardware accelerators to provide very fast simulation speeds. FAST was designed with a heavy emphasis on simulation speed and tight coupling between the front-end, back-end, and FPGA accelerator, and because of this has a different scope than QSim.

Coremu [19] is an emulation project that runs the CPUs of a modified QEMU in parallel with emphasis on performance. Techniques used to achieve fast parallel execution on Coremu could be ported to QSim to improve its multi-threaded performance, overcoming the major shortcoming of Coremu from a simulation perspective, that it lacks any way to instrument or control the execution of the emulated code.

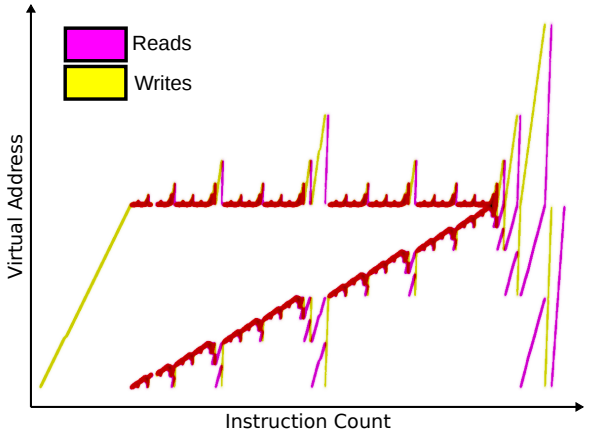

Figure 8: Plot of memory accesses to array and temporary copy area during serial merge sort. Yellow represents writes and magenta represents reads.

Pin [11] and Shade [6] are binary instrumentation utilities, and at least Pin has been used as front-end for simulation [9, 13], but neither provide support for emulating operating system code. There are other functional simulation products specifically designed to be used as part of larger simulation frameworks [12, 3, 2], and there have been successful simulation infrastructures built using these [10, 1]. What has not been created until now is a unified instruction set independent API through which a set of potentially parallel back-ends can be interfaced with a set of front-ends with considerably less effort than required by current front-ends.

Some of the aforementioned front-ends, including Coremu, Simics, Simnow, and Simplescalar are distinct from QSim in that they provide *full system* emulation, providing a full range of hardware, including network and disk devices. While such devices can be built on top of QSim's callback interface, none are currently available, limiting QSim's focus to machine models that consist solely of processors and RAM. Despite this limitation, QSim still allows more simulation fidelity than front-ends based on Pin, Shade, or other user-mode-only front-ends. Even though it does not emulate the full set of I/O devices provided by QEMU, operating system code is still emulated, including filesystem, memory management, and the scheduler.

7. PERFORMANCE AND SCALABILITY

Table 2 shows the slowdown for a single-threaded instance of the QSim emulator running on our test machine with all callbacks set to empty functions. This is an estimate of the speed of QSim when it is being used as a simulator front-end which sets all callbacks. From this we gather that typical slow-down over native execution of 4-threaded applications on four CPUs is on the order of 300.

Scaling of QSim with empty callbacks up to four threads is seen in Figure 9. This highlights one of the limitations of

Program	Data Set	Description	Slowdown*	MIPS
swaptions	102400 simulations, 4 swaptions	Options pricing simulation from Parsec benchmark suite.	259x	18.5
mtgl-bfs	2^{14} vertex RMAT graph	Multi-Threaded Graph Library RMAT generation and breadth-first search.	387x	36.6
ocean-non-contig	258x258 grid	Ocean from Splash-II benchmark suite.	267x	40.7

*Slowdown over native with all callbacks set to empty functions.

Table 2: Programs used to test QSim performance and scalability, running in four host threads. Average emulation speed is 24.07 MIPS.

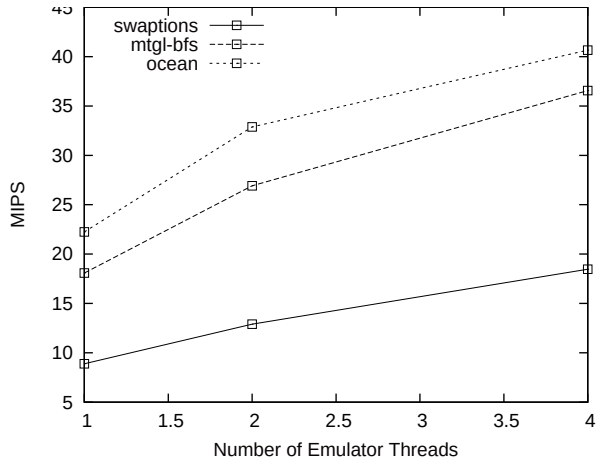

Figure 9: Performance as QSim scales to multiple threads, measured using a set of common parallel benchmarks.

the QSim implementation: atomic memory operations must wait for all currently running threads to either encounter an atomic operation or finish running before they can begin, often causing large delays when synchronization activity is bursty. This is most pronounced when the number of instructions passed to `run()` is large relative to the typical distance between atomic memory operations. For the evaluation, this argument to `run()` is always 1000 instructions.

Large numbers of guest threads are also well-supported, the execution rate tends to be the same no matter how many QEMU CPUs are instantiated beyond the number of hardware threads provided by the host system, though the per-cpu execution rate falls roughly linearly. This means that, on a single-core host, running a single instruction on each of a thousand guest cores will require the roughly the same time amount of time as running a thousand instructions on a single guest core.

8. FUTURE WORK

Future work is primarily concerned with creating fast and flexible simulators using QSim as a front-end. An implementation of the QSim API that communicates with the emulator over a network interface has been created to allow exploration of parallel and distributed simulators. The current implementation is expected to communicate with a set of nodes in a parallel simulation which uses MPI or a similar framework. All of the nodes in the simulator would communicate out-of-band with Remote QSim through a client-server interface.

Other future work involves porting QSim to other instruction sets and using the QSim API for other front-ends. 64-bit x86 support is nearly complete and a port to ARM using QEMU's ARM support has just commenced.

9. CONCLUSIONS

Features of QSim enabling its use as an effective simulation front-end include:

- Enables execution driven simulation of multithreaded processors.
- Allows parallel execution in both the front-end and back-end of the simulator.
- Supports hundreds of guest cores, providing capability to model the next generation of manycore processors.
- Runs operating system as well as user code, providing insight into systems software.
- Provides an instruction set independent API, allowing construction of universal simulators.
- Supplies tool for saving and loading state, enabling checkpointing and pre-fastforwarding.

QSim has demonstrated a way to build front-ends for processor simulators for multicore microarchitecture research, enabling parallel execution-driven simulators. Through thread-safety in the API and parallel execution in the emulator itself, QSim enables parallel simulators. The QSim API additionally provides a way to combine front-ends and back-ends without the effort of starting over each time. This has already enabled the creation of an instruction set independent processor simulator, which can be either trace or execution driven, and more are expected in the future.

10. ACKNOWLEDGMENTS

The authors are grateful to Paolo Faraboschi and Daniel Ortega for their suggestions and guidance in getting QSim started. This work was supported by the National Science Foundation under grant CNS855110, Sandia National Laboratories, and HP Laboratories.

11. REFERENCES

[1] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. Cotson: infrastructure for full system simulation. *ACM SIGOPS Operating Systems Review*, 43(1):52–61, 2009.

[2] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[3] R. Bedichek. Simnow: Fast platform simulation purely in software. In *Hot Chips*, volume 16, 2004.

[4] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[5] D. Chiou, D. Sunwoo, J. Kim, N. Patil, W. Reinhart, D. Johnson, J. Keefe, and H. Angepat. Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators. In *Proceedings of the 40th Annual IEEE/ACM international Symposium on Microarchitecture*, pages 249–261. IEEE Computer Society, 2007.

[6] B. Cmelik and D. Keppel. *Shade: A fast instruction-set simulator for execution profiling*, volume 22. ACM, 1994.

[7] G. Dabah. distorm64 - the ultimate disassembler library., 2009.

[8] R. M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, 1990.

[9] A. Jaleel, R. Cohn, C. Luk, and B. Jacob. Cmp$im: A pin-based on-the-fly multi-core cache simulator. In *Proc. of the The Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, pages 28–36, 2008.

[10] G. Loh, S. Subramaniam, and Y. Xie. Zesto: A cycle-level simulator for highly detailed microarchitecture exploration. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 53–64. IEEE, 2009.

[11] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200. ACM, 2005.

[12] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, pages 50–58, 2002.

[13] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.

[14] A. Patel, F. Afram, and K. Ghose. Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors. In *1 st International QEMU Users' Forum*, page 29, 2011.

[15] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramirez, and M. Valero. Trace-driven simulation of multithreaded applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 87–96, Washington, DC, USA, 2011. IEEE Computer Society.

[16] A. Rodrigues, K. Hemmert, B. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, et al. The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):37–42, 2011.

[17] J. Shin, D. Huang, B. Petrick, C. Hwang, A. Leon, and A. Strong. A 40nm 16-core 128-thread sparc® soc processor. In *Solid State Circuits Conference (A-SSCC), 2010 IEEE Asian*, pages 1–4. IEEE, 2011.

[18] R. Singhal. Inside intel® next generation nehalem microarchitecture. In *Hot Chips*, volume 20.

[19] Z. Wang, R. Liu, Y. Chen, X. Wu, H. Chen, and B. Zang. Coremu: a scalable and portable parallel full-system emulator. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, pages 213–222. Citeseer, 2011.

[20] M. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. *Performance Analysis of Systems and Software, IEEE International Symmposium on*, 0:23–34, 2007.