# Boosting Instruction Set Simulator Performance with Parallel Block Optimisation and Replacement

**Brad Alexander**[1] **Sean Donnellan**[1] **Andrew Jeffries**[3] **Travis Olds**[3]
**Nicholas Sizer**[1]

[1] School of Computer Science,
University of Adelaide,
Adelaide 5005,
Email: brad@adelaide.edu.au

[2] Ultra Electronics Avalon Systems,
12 Douglas Drive,
Mawson Lakes,South Australia 5095,
Email: andrewj@avalon.com.au

[3] Australian Semiconductor Technology
Company,
Level 5, 76 Waymouth Street,
Adelaide, South Australia 5000,
Email: travis.olds@astc-design.com

## Abstract

Time-to-market is a critical factor in the commercial success of new consumer devices. To minimise delays, system developers and third party software vendors must be able to test their applications before the hardware platform becomes available. Instruction Set Simulators (ISS's) underpin this early development by emulating new platforms on ordinary desktop machines. As target platforms become faster the performance demands on ISS's become greater. A key challenge is to leverage available simulator technology to produce, at low cost, incremental performance gains needed to keep up with these demands. In this work we use a very simple strategy: in-place-block-replacement to produce improvements in the performance of the popular QEMU functional simulator. The replacement blocks are generated at runtime using the LLVM JIT running on spare processor cores. This strategy provides a very lightweight way to incrementally build an alternate code generator within an existing ISS framework without incurring a substantial runtime cost. We show the approach is effective in reducing the runtimes of the QEMU user-space emulator on a number of SPECint 2006 benchmarks.
*Keywords: Instruction Set Simulation, Dynamic Binary Translation, Background Optimisation, LLVM, QEMU*

## 1 Introduction

Instruction Set Simulators (ISSs) are software platforms that run on a host hardware architecture and emulate a guest hardware architectures. An ISS allows developers to test and use systems and application software whenever using the actual hardware platform is not an easy option. ISSs are used for reasons of security and safety[Vachharajani et al., 2004], cross-platform-support[Adams and Agesen, 2006, Bellard, 2005, Chernoff et al., 1998] or just because they may be more readily available than the actual hardware. In the extreme, an ISS can provide a platform for software development *before* the corresponding hardware platform exists. This last mode of use for ISSs permits the parallel-development of hardware and software for new mobile and embedded devices. Such parallel-development reduces time-to-market which is vital in the commercial success of these devices[De Michell and Gupta, 1997]. With mobile devices becoming faster, and competitive pressures shortening production cycles, there is strong demand for faster emulation from ISSs.

ISSs emulate at a variety of levels. Cycle-accurate ISSs[Lee et al., 2008] emulate the timing of hardware devices to allow debugging at the hardware/system interface. Functional ISSs[Adams and Agesen, 2006, Bellard, 2005, Magnusson et al., 2002, Cmelik and Keppel, 1994, Witchel and Rosenblum, 1996] emulate architectures at the behavioural level providing a platform for interactive testing of systems and application software. Functional ISSs are generally much faster than cycle-accurate ISSs making them attractive for high-level system and application developers. Functional ISSs can further subdivided into whole-system simulators, able to emulate an entire operating system[Adams and Agesen, 2006, Bellard, 2005, OVP, 2011] and process-VMs or user-mode emulators which can run a single application. This work describes enhancements to the performance of the QEMU ISS[Bellard, 2005]. QEMU is one of the most popular ISS's. It provides fast emulation for a variety of target and source architecture and forms the basis of a number of industrial emulators including the Android mobile device emulator[Google .Inc, 2011]. QEMU provides both system-mode emulation for whole-systems and user-mode emulation for single-applications. In this article we focus on enhancements to the simpler, and faster, user-mode QEMU.

Like many ISSs, QEMU uses Dynamic Binary Translation (DBT) to translate blocks of guest platform code to host platform code at runtime. Once blocks are translated, they can be run natively on the host platform – greatly improving performance over simple interpretation of guest instructions[Arnold et al., 2005]. However, good performance is only possible if translation is done quickly. On single processor systems, any attempt at optimising translated code is time-constrained: time spent optimising translated code is time *not* spent running translated code. When emulating on a multi-core host these constraints are

Copyright ©2012, Australian Computer Society, Inc. This paper appeared at the 35th Australasian Computer Science Conference (ACSC 2012), Melbourne, Australia, January-February 2012. Conferences in Research and Practice in Information Technology (CRPIT), Vol. 122, Mark Reynolds and Bruce Thomas, Ed. Reproduction for academic, not-for-profit purposes permitted provided this text is included.

less severe. After an initial fast translation, idle cores can be utilised to perform background optimisation in a separate thread. Such background optimisation is relatively common in high-level-language process-VMs[Krintz et al., 2001, Alpern et al., 2005] but relatively rare in DBT-Based ISSs[Qin et al., 2006]. For simulators where the host and guest Instruction-Set-Architecture (ISA) is the same, this rarity is unsurprising: we would expect gains from further optimisation of already-optimised code in same-ISA simulation to be small[Adams and Agesen, 2006, Arnold et al., 2005]. However, this rarity is harder to explain in different-ISA ISSs, the architectural gap between the guest ISA and the host ISA is usually large enough to leave room for further optimisation.

In this article we describe an implementation that experiments with the use of background optimisation to improve the speed of user-mode QEMU running ARM binaries on a multi-core x86-64 host. In our experiments we use LLVM components, running on a spare core, to perform background optimisation and replacement of cached blocks of dynamically translated code. We show that this approach is successful in producing speedups in the simulation of a number of the SPECint 2006[Standard-Performance-Evaluation-Corporation, 2011] benchmarks.

At this point it should be emphasised that the focus of this work is in utilising spare cores to improve code quality and hence the simulation speed of *each-thread* of an application. This is in contrast to the orthogonal, and well-studied, problem of using multiple cores to help simulate *multi-threaded* applications[Jiang et al., 2009, Almer et al., 2011, Wentzlaff and Agarwal, 2006]. The primary aim of this work is to improve thread code quality with the goal of reducing overall execution time.

This paper makes the following contributions:

- It is the first successful attempt to boost the performance of QEMU through background optimisation.

- Moreover, to the best of our knowledge, it is the second successful attempt to use background optimisation of already translated code to boost the performance of any different-ISA ISS, the other being SIMIT-ARM[Qin et al., 2006] and the first to use spare cores on the same machine.

- It demonstrates that background optimisation can be added cheaply and incrementally and still achieve faster execution (section 4.5) and better quality host-code (section 4.4). Performance gains can be had *before*, covering all instructions instructions and before implementing heuristics for selecting blocks for replacement. Moreover, our optimisation is still useful at the level of the basic block - allowing gains to be enjoyed before block aggregation and inter-block optimisation is implemented. In summary, there is a path to retro-fit changes cheaply, incrementally and productively to an existing ISS, using standard tools.

The rest of this paper is structured as follows. In the next section we describe related work. In section 3 we describe our approach starting with the base components of QEMU and LLVM followed by an explanation how these are combined to make Augmented-QEMU which uses background optimisation for faster execution. In section 4 we describe our experimental results. Finally, in section 5 we summarise our results and propose future directions for research.

## 2 Related Work

The most closely related work to ours is Scheller's LLVM-QEMU project[Scheller, 2008]. Like our work, LLVM-QEMU used the LLVM Just-in-Time compiler (LLVM-JIT). However, in his work he uses the LLVM-JIT to *replace* the role QEMU's native code generator – the Tiny Code Generator (TCG). In our work we the LLVM-JIT to *complement* the role of TCG. We allow TCG to perform a fast initial translation for immediate execution and use the LLVM-JIT to replace selected blocks as QEMU is running. Our work also differs in the use of a separate thread[1] to run the LLVM-JIT. This combination of differences results in our implementation achieving faster performance overall with speedup, rather than slowdown, on most benchmarks.

Also related is the identically named LLVM-QEMU project by Chipounov and Candea[Chipounov and Candea, 2010]. They also used the LLVM-JIT to bypass TCG in QEMU. They compiled all micro-operations into C functions which were then translated into LLVM Intermediate Representation (LLVM IR) and then optimised before execution. Again, the overheads of running the LLVM-JIT were substantial, resulting in slowdown when compared to the original QEMU.

Again, our work differs most from the above approaches in leaving TCG intact. By leaving most of the QEMU infrastructure in place we are able to focus on incrementally increasing the number of blocks we handle as we add new, carefully-handwritten, translations from individual ARM instructions to LLVM-IR code. These direct hand-written translations appear, a-priori, easier to optimise than LLVM-IR generated from C or TCG intermediate code.

Looking more broadly at related DBTs, same-ISA DBTs[Watson, 2008, Adams and Agesen, 2006] have the option of preserving most optimisations already present in the source binary. This can give very good performance[Adams and Agesen, 2006] though, in some cases, knowledge gained from runtime profiles can be used for even more optimisation[Bruening et al., 2003]. Unfortunately most ISSs are by necessity different-ISA DBTs[OVP, 2011, Magnusson et al., 2002, Qin et al., 2006, Bellard, 2005]. These face greater challenges than same-ISA DBTs due to the unavoidable loss of platform-specific optimisations during translation between ISAs.

The trade-off between running translated code and optimising translated code means that optimisation must be done sparingly on a single processor machine. On multi-core machines, optimisation can be carried out in a separate thread. Such background optimisation relatively common in process VMs for high level languages such as Java (Kulkarni[Kulkarni et al., 2007] gives an overview of the impact of these optimisations). In contrast, the use of such background optimisation is rare in different-ISA DBTs.

One exception is SIMIT-ARM[Qin et al., 2006] which, concurrently with interpretation of guest code, translates guest code into large, fixed-size blocks of C code and then compiles these into dynamically-linked-libraries (DLLs) using gcc running on separate hosts. As the new DLLs are produced they are loaded in and run. The amount of optimisation performed by gcc is trivially controlled using compiler flags. The DLLs in SIMIT-ARM are persistent between runs which lets applications run much faster the second time they are invoked. In earlier experiments[Lee, 2009] it was

[1]It should be noted that LLVM-QEMU was a short summer-project and multi-threading was on the list of things to do.

found that SIMIT-ARM was able to run at speeds comparable to QEMU in user-mode when it runs on these cached DLLs.

Our work differs from SIMIT-ARM in our use of spare cores rather than separate hosts, and the focus of optimisation on small basic blocks rather than large fixed-sized blocks with multiple entry points[2]. Our approach, through the use of a lightweight-JIT, working in a shared memory space, provides a lower latency on block optimisation – which is likely to be important in the context of the relatively fast QEMU simulator.

Finally, there is substantial work that applies optimisation to the *initial* translation phase of an ISS. Almer [Almer et al., 2011] uses the LLVM-JIT to translate and optimise hot traces in their multi-threaded ISS. Wentzlaff[Wentzlaff and Agarwal, 2006] also performs translation and optimisation of code blocks in their parallel ISS. Both these works differ from ours by applying optimisation only during the initial translation of each trace or block as such they do not have to perform runtime block-replacement of translated code.

This concludes our overview of related literature. In the next section we describe our approach.

## 3 Approach

In this section we describe the implementation used in this work: Augmented-QEMU. The goal of Augmented-QEMU is to provide faster emulation by performing background optimisation of guest (ARMv5) instructions to host (x86-64) instructions. Augmented-QEMU is built using the following software components:

- QEMU version 0.10.6, running in user-mode, which we call vanilla-QEMU,
- The LLVM optimisers and x86 code-generator that are part of the LLVM just-in-time compiler version 2.6 (LLVM-JIT).

We describe each of these in turn before explaining how they are combined to form Augmented-QEMU in section 3.3.

### 3.1 Vanilla-QEMU

QEMU[Bellard, 2005] is a widely used, versatile and portable DBT system. QEMU is cross-platform. It supports a variety of host and guest ISAs. It is able to exploit the features of its host architecture. For example, when run on a 64bit host such as the x86-64 architecture it is able to exploit the extra registers to provide better performance. QEMU is a functional simulator, it emulates program behaviour rather than simulating accurate timings of events in hardware. QEMU has two modes: a system-mode with detailed hardware models for emulating entire systems; and a, somewhat faster, user-mode that acts as a process-VM on which to run a single application. QEMU is quite fast, with quoted emulation speeds of 400 to 500 MIPS. In terms of actual runtimes, on our experimental machine, we found that QEMU running cross-compiled ARM binaries ran approximately ten times slower than native x86 compiled benchmarks. The following is a brief outline of how QEMU CPU simulation works. For a more detailed overview see [Bellard, 2005].

[2]The approach of SIMIT-ARM is highly original and starkly different from the basic-block-oriented approach taken by most DBTs. The use of multiple-entry points leads to blocks being cast as large switch statements. The potential impact of this format on different levels of optimisation is unknown and warrants further study.

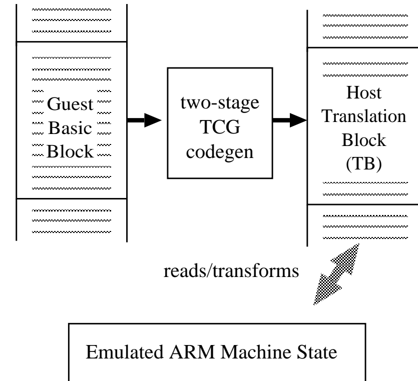


Figure 1: Basic Translation Schema for QEMU

#### 3.1.1 The QEMU Simulation Process

QEMU performs computation by maintaining an abstract CPU state for the guest architecture. This state includes program counters, stack pointers, control flags and general-purpose registers. QEMU translates guest binary code to host code that then acts on this processor state. At the start of simulation, this state is initialised as it would be at the start of program execution on the guest architecture. For Augmented-QEMU, the guest architecture is a 32 bit ARM machine and the state is called `CPUARMState`. Figure 1 illustrates this basic execution schema. Note that, unlike some other VMs[Bruening et al., 2003, Krintz et al., 2001], QEMU does not initially directly interpret guest code. Instead it relies on a fast translator, called Tiny Code Generator (TCG) to quickly produce basic blocks of host code. These blocks, called Translation Blocks (TBs), are cached in an area called the translation cache and then immediately executed. TCG runs in two-phases. The first phase translates the guest ISA to TCG code – a generalised intermediate form. The second phase converts TCG code to the host ISA. This two-phase design aids portability by decoupling the source and host ends of the translation process.

The translation schema described above sits in the context of the broader control structure for QEMU shown in figure 2. In the figure, control flows are indicated with thin lines and flows of code with thick dashed grey lines. At the core of QEMU simulation is the `cpu_exec()` function. This function is responsible for the controlling the translation and execution of basic blocks of guest code. During simulation, `cpu_exec()` operates as follows. First, when a block of guest code needs to be executed, the source program counter (SPC) is read from the guest CPU state (step (1) in figure 2). Next, the SPC is looked up in the map table (step (2)). At this point one of two things can happen:

1. The block starting at the SPC is found to have already been translated and stored in the translation cache. In this case the map table will return the address of the relevant TB in the translation cache. Or,
2. The block starting at the SPC is not in the translation cache and so is not found in the map table. In this case the guest block is new and will need to be translated prior to execution.

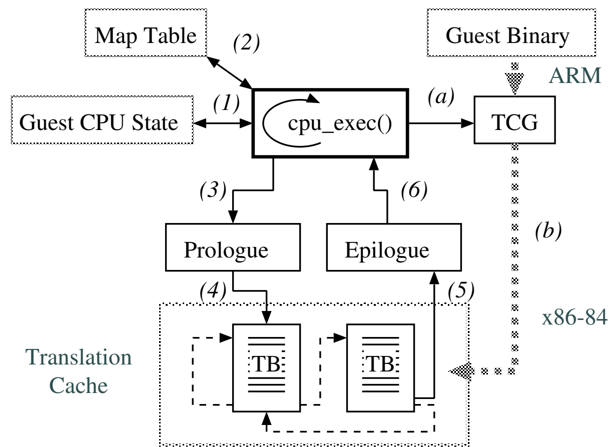


Figure 2: Control structure for QEMU centred around the core execution loop: `cpu_exec()`. Showing the actions creating new TBs ((a) and (b)) and actions to execute extant TB's ((1) to (7)).

In the case of the relevant TB being found, control is passed from `cpu_exec()` to a block of prologue code (step (3)). The prologue saves QEMU's register state before jumping to the relevant TB (step (4)). The TB then executes. In many cases, when the end of the TB is reached, control is not passed back to `cpu_exec()` but instead the TB branches to itself or other TBs. This internal branching within the translation cache (indicated with thin dashed lines in in figure 2) is called *chaining*. Chaining is a simple optimisation that often avoids the overhead of going back to `cpu_exec()`. QEMU is able to spend substantial time running in chained TBs of blocks within the translation cache. Eventually, one of the TBs will return control back to some epilogue code (step (5)) which updates the SPC in the guest CPU state and and restores the QEMU register state before returning to `cpu_exec()` (step (6)).

In the case of the required TB *not* being found, a new TB will have to be generated from the guest binary. This is done by calling TCG with the current SPC (step (a)) and inserting it into the translation cache (step (b)). Note that the chaining between TBs is performed by TCG as blocks are inserted into the translation cache. Once the new TB is in the cache, an entry for it is inserted into the map table and its execution can proceed.

It should be noted that, overall, this simulation infrastructure is quite efficient, in earlier experiments[Jeffery, 2010] we found that QEMU spent a large majority of its time running in the translated code cache when run in user-mode on SPECint 2006 benchmarks.

This concludes our brief overview of vanilla QEMU. It is worth noting that our modifications, which we describe shortly, leave this basic structure intact. Our changes basically leverage LLVM-JIT described next, to improve the code in the TBs.

### 3.2 LLVM

The LLVM compiler Infrastructure[Lattner and Adve, 2004] is a set of open-source components which can be used as building blocks for custom compilers. The aim of the LLVM project is to provide modular components, such as optimisers, and code-generators that can be reused in different language implementations. The key to reusability is the use of a general purpose intermediate representation, LLVM-IR, to act as an interface between components. The LLVM project envisages that any compiler using LLVM components can have easy access to the frequent improvements made to these components.

LLVM is well-supported and being used in many significant projects[LLVM-Project, 2011]. In this work, we combine a short series of optimisation passes, described in the section below and the LLVM-JIT code generator to produce blocks of fast x86-64 code from LLVM-IR that our own custom translator generates from ARM instructions. Next, we describe the process by which we exploit these components in Augmented-QEMU.

### 3.3 Augmented-QEMU

Augmented QEMU performs background optimisation and replacement of blocks, using LLVM, to improve the performance of QEMU. The modifications used to produce Augmented-QEMU in the context of vanilla-QEMU are shown in figure 3. Note, to provide context, the components of vanilla-QEMU from figure 2 are shown in grey. The new and modified components are drawn in black. Control flows are represented by solid black arrows and data flows (labelled with their type) by dashed grey arrows. There are two threads, the original QEMU thread and a new LLVM thread. Communication between the threads is managed by two queues, the block translation queue, which contains pointers to guest binary blocks awaiting further optimisation; and the block replacement queue, which contains pointers to optimised replacement TBs. A brief summary of how Augmented QEMU works follows. Each part of the summary is cross-referenced to the steps in figure 3. In addition, more detailed descriptions of these parts follow this summary.

**Steps (i) and (ii), Testing Eligibility:** Whenever TCG produces a new TB from a block of guest binary code, our modified `cpu_exec` performs an eligibility check. The eligibility check scans the block of guest binary and checks that we have implemented LLVM translations for every instruction in the block (step (i)). If so, a pointer to this block is queued for translation (step (ii)). This process is described in section 3.3.1.

**Steps (iii) and (iv) , Block Translation:** The addition of a new pointer to the block translation queue wakes the LLVM thread (step (iii)) which immediately passes the block of guest binary to our own LLVM-IR code generator for translation (step (iv)). This process is described in section 3.3.2.

**Steps (v) and (vi), Block-optimisation/Code Generation:** The new block of LLVM IR is given to the LLVM-JIT (step (v)). The LLVM-JIT performs a series of optimisations and then generates a block of host code. This process is described in section 3.3.3. A pointer to this new code is added to the block replacement queue (step (vi)).

**Steps (vii),(viii) and (ix), Block-Replacement:** The replace-block function, de-queues pointers to any newly generated blocks (step (vii)) and, if the block fits in the space allocated by the original TB, it uses a memcopy operation (step (viii)) to overwrite the original TB block. Finally, a jump instruction is added from the end of the new code block to the end of the space occupied by the original

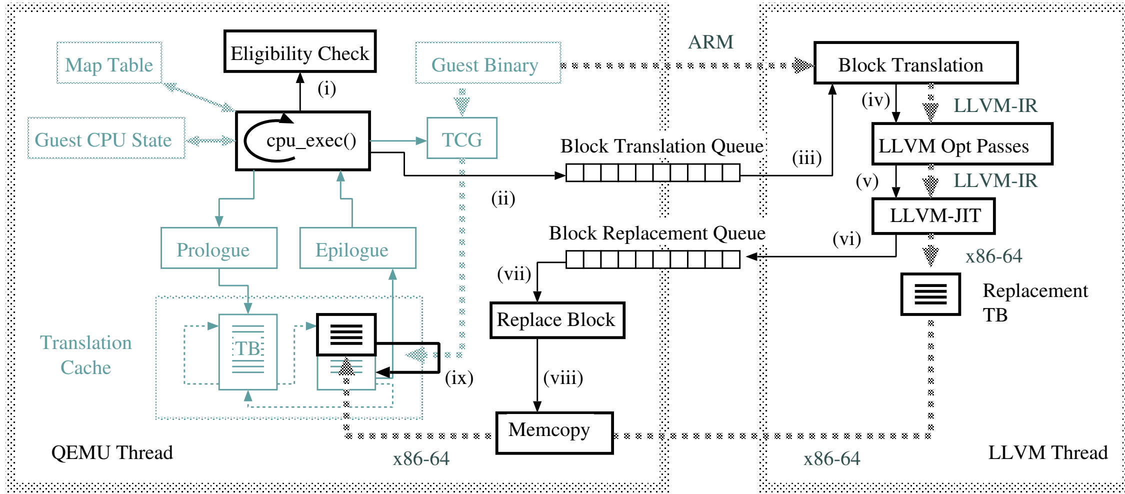


Figure 3: The modifications used to produce Augmented-QEMU (in black) in the context of vanilla-QEMU (in grey).

| Execution Count | |
|---|---|
| seng | |
| ldr | 2594556723 |
| add | 618512755 |
| str | 595786363 |
| sub | 432138506 |
| cmp | 428270933 |
| mov | 408249153 |
| bne | 204144055 |
| lsl | 170450711 |
| beq | 96700869 |
| bl | 94330545 |
| asr | 87404495 |
| bx | 63627453 |
| and | 58944875 |

Table 1: Top instruction execution counts for seng SPECint 2006 benchmark (instructions we translate are in green)

TB (step (ix)). This process is described in section 3.3.4

More detailed explanations of each of these steps follow.

### 3.3.1 Testing Eligibility

Augmented-QEMU uses a slightly modified version of `cpu_exec()` to guide the block-optimisation and replacement process. Immediately before any new TB is formed by TCG, an eligibility check is performed on the same block of guest binary code to determine if it can also be translated by the LLVM thread. This check needs to be done because not all of the guest binary instruction set is handled by our LLVM optimisation thread. The choice of which instructions to handle first is guided by benchmarking. We ran a number of the SPECint 2006 benchmarks compiled to ARM by GCC 4.4.3 (arm-softfloat-linux-gnueabi) to get counts of each instruction. The counts for the sjeng chess-playing benchmark on test workload are shown in table 1. The instructions marked in green are among the ones currently translated by our block translator[3]. Branches, marked in yellow, are excluded because they appear only after the end of each translation block, either as part of chaining or jumps back to the epilogue. Instructions in red are yet to be implemented in the block translator. Other SPECint 2006 benchmarks had roughly similar distributions. In our tests, data movement, arithmetic and comparison operators tended to dominate guest-code so we focused on implementing these. Note that only blocks that we can translate *entirely* are considered eligible. By this measure, currently more than half the total number of blocks in the SPECint 2006 benchmarks are eligible.

After a guest block is found to be eligible for translation, a pointer to that block is queued for translation by the LLVM thread. Access to the translation queue is guarded by a lock to prevent race-conditions. No lock is needed for accessing the guest binary code because it is read-only in this context[4].

### 3.3.2 Block Translation

The addition of the new block-pointer to the queue wakes up the LLVM thread which de-queues the pointer and gives it to the block translator. This stage generates LLVM-IR for each ARM instruction in the block in the guest binary referenced by the de-queued pointer. The block translator is hand-written by us and its creation was the most labour-intensive part of this project. This block translator is, essentially, a partially complete ARM binary to LLVM-IR front-end for the QEMU virtual machine. In detail, the block-translation:

1. Sets up a function prototype with a call to `LLVMFunctionType()`, so that a pointer to the guest CPU state (`CPUARMState`) structure is passed as the first parameter making it accessible to the code generated by LLVM.

2. Allocates a new empty LLVM IR function using a call to `LLVMAddFunction()` with the prototype created in step 1 above.

[3] We also implemented translations for a number of logic operators such as `eor` and `orr` not on the list above.

[4] Any attempt at self-modification by the guest-binary causes QEMU to flush it's entire translation cache and map-table.

```
define { [16 x i32] }* @block_test({ [16 x i32] }*) {
entry:
 %tmp1 = getelementptr { [16 x i32] }* %0, i32 0, i32 0, i32 0
 %tmp2 = getelementptr { [16 x i32] }* %0, i32 0, i32 0, i32 2
 %tmp3 = load i32* %tmp2
 store i32 %tmp3, i32* %tmp1
 ret { [16 x i32] }* %0
}
```

Figure 4: LLVM intermediate representation of `mov r2, r0`

3. Transcribes the ARM instructions one at a time with the help of an LLVM builder object made by a call to `LLVMCreateBuilder()` on the function described in step 2 above. Using this builder LLVM IR instructions can be added to the function by calling functions such as: `LLVMBuildAdd()`, `LLVMBuildStore()`, `LLVMBuildIntToPtr()` and several others.
4. Finally, finishes off the function with a call to `LLVMBuildRet()` to build a return statement.

At the end of this process we have a pointer to the new LLVM IR function.

To help visualise block-translation, consider a hypothetical block consisting of a single ARM instruction[5]: `mov r2, r0`. Our Block Translator would build the LLVM IR function shown in figure 4.

A short semantics of this code is as follows. `%tmp1` holds a pointer into state pointing to `r0`, while `%tmp2` holds a pointer to `r2`. The value at the address in `%tmp2` is then loaded into `%tmp3`, which is subsequently stored at the location pointed to by `%tmp1`. At this stage our `mov` instruction is complete and the function returns.

Note that each basic block is mapped by our block translator to one LLVM-IR function. Each LLVM-IR function created by the block translator is immediately passed to the optimisation and code generation stages. We describe these stages next.

#### 3.3.3 Code optimisation and generation

These stages are calls to pre-existing LLVM compiler components. Thus, from the implementer's point of view, this stage is straightforward. We first add optimisation passes to the LLVM JIT and then call the LLVM JIT to optimise and generate the LLVM IR block created by the block translator described in the previous section. In this project, we add optimisation passes by calling the following functions:

```
LLVMAddConstantPropagationPass();
LLVMAddPromoteMemoryToRegisterPass();
LLVMAddDeadInstEliminationPass();
LLVMAddDeadStoreEliminationPass();
LLVMAddInstructionCombiningPass();
LLVMAddGVNPass();
```

The purpose of each of these passes is fairly self-explanatory except for the `LLVMAddGVNPass` which does global value numbering to help eliminate equivalent values.

Once the passes are set up, then the LLVM JIT can be called on each block to perform translation. This translation can be triggered simply by calling `LLVMGetPointerToGlobal()` which returns a pointer to the optimised and translated host (x86-64) block. The host block produced from the code in figure 4 is shown in figure 5. This code bears some explana-

```
mov %r14, %rdi
mov 0x8(%rdi), %eax
mov %eax, (%rdi)
mov %rdi, %rax
ret
```

Figure 5: The x86-64 block produced by the LLVM JIT for the code shown in figure 4.

tion as the first instruction is not in-fact generated by LLVM. `mov %r14, %rdi` is related to the LLVM function prototype outlined previously: The x86-64 C ABI designates that the first struct pointer provided as a parameter to a function should be passed in register `%rdi`, however QEMU pins the CPU state pointer in `%r14`. As LLVM doesnt support pinned registers[6] the `mov` instruction is introduced as a work-around to move the CPU state pointer to `%rdi` where it can be accessed by the LLVM-generated code. The next two lines implement the actual move and the last two lines return the pointer to the register state. These last two lines are not required when the the code is copied back to the TB during block replacement so they are dropped during this process.

Once the x86 block has been generated as above, it is added to the block-replacement queue. We describe how block replacement works next.

#### 3.3.4 Block Replacement

The task of block replacement is to copy the host code blocks made by the LLVM JIT back into the correct TBs in the translation cache. Block replacement has to be carefully timed so as not to overwrite the contents of a block while it is running. In our implementation we took the simple choice of only allowing block replacement just before TCG is called to translate another new block. This choice guarantees safety – QEMU is guaranteed not to be running in the code cache when it is about to invoke TCG. Moreover, as we shall see in section 4.2 it allows for tolerably fast block replacement.

Block replacement is done by a simple memcopy back to the address of the original TB. The block replacing the original TB is usually shorter than the original TB. When it is shorter, we add code at the end of our new block to jump to the chaining section of the original TB (step (ix) of figure 3). In rarer cases where the optimised block is too big to fit back into the original TB we simply discard the new block on the assumption that, being so large, it is unlikely to be efficient in any case.

There is one more important detail to add. We had to make a special arrangement for dealing with the ARM compare instruction: `cmp`. This instruction is not easily implemented efficiently in TCG intermediate code so, instead, QEMU uses a pre-compiled

[5] We actually do get a few such one-instruction blocks in our benchmarks.

[6] which would have allowed us to use `%r14` directly

| bench | total | queued | gen | repl | discard |
|---|---|---|---|---|---|
| hmmer | 5231 | 2417 | 2413 | 2297 | 116 |
| gcc | 69411 | 40912 | 40886 | 37461 | 3425 |
| sjeng | 4717 | 2529 | 2525 | 2460 | 65 |

Table 2: Comparison of blocks queued, generated, replaced and discarded compared to the total number of blocks in three SPECint 2006 benchmarks.

helper function to implement `cmps` semantics. However, when using the LLVM-JIT, by far the easiest option is to *inline* the x86 code for `cmp`. This adds, in the worst case, 48 bytes of extra space to our compiled block. To account for this we added a small amount of logic to the QEMU code that allocates space for each new block in its code cache. This logic allocates 48 bytes of additional space in each new TB to allow for a bigger inlined version of the TB to be copied back in.

This concludes our description of the modifications made to implement augmented QEMU. Next we assess the impact of these modifications on QEMU performance.

## 4 Results

This section presents our experimental results. All experiments were carried out on an Intel Core 2 Quad Processor Q8200, running at 2.3GHz, with 4GB of memory. This is an x86- 64 architecture providing, more general purpose registers than IA- 32 architectures.

We summarise our results in terms of code-coverage (next), measured code size (section 4.3),timeliness of block-replacement 4.2, informal measures of code quality (section 4.4), and, importantly, code speed (section 4.5). We discuss each of these in turn.

### 4.1 Code Coverage

Code coverage is the number of blocks we are able to replace in our benchmarks. In Augmented QEMU this is, primarily, a function of choice and number of instructions supported by our block translator. We found our code coverage to be quite consistent across benchmarks. Table 2 presents statistics collected for three SPECint 2006 benchmarks running on test data. As can be seen in all three applications slightly more than half the total number of basic blocks were found to be eligible for block-translation and queued. Of these, most had time to be generated and most of these were small enough to replace the original block. A very small percentage were discarded because they were too large - perhaps indicating reasonably effective optimisation. In summary, Augmented QEMU, is able to achieve moderately good code coverage with a small number of implemented instructions. A very high percentage of eligible blocks go on to be replaced.

### 4.2 Timeliness of Replacement

As we can see from above, a reasonable number of blocks are being replaced on the benchmarks. However, these blocks will not do much good if they are not being replaced in a timely manner.

To assess the speed of block replacement we plotted a count of blocks queued and blocks replaced over time during the running of the sjeng benchmark. Figure 6 shows the blocks queued (blue dots) and the blocks replaced (red dots) over the entire length of the sjeng benchmark on the test workload. As can be seen in this benchmark, queueing and replacement of blocks track each other closely. Moreover the great majority of blocks are replaced quickly. This fast replacement helps maximise their chance of being run. Note that a small gap grows between blocks queued and blocks generated as the run proceeds. This gap is due to the small number of blocks being discarded for being too large. Our manual tracking of blocks queued and replaced on other benchmarks revealed a similar pattern to above.

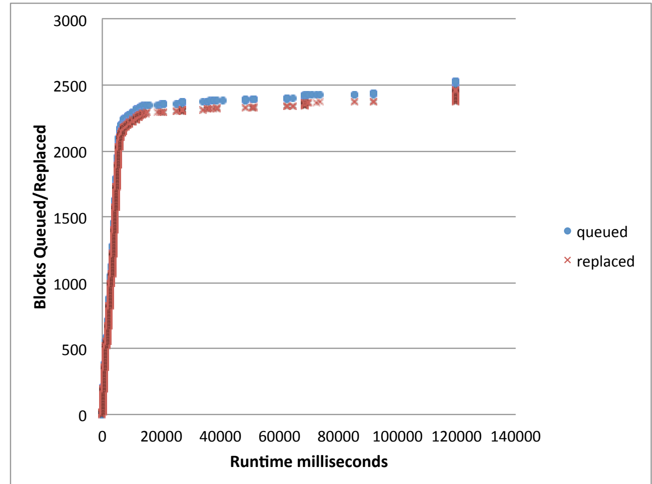


Figure 6: Cumulative block totals for queueing (blue dots) and replacement (red dots) over length of sjeng benchmark on test workload

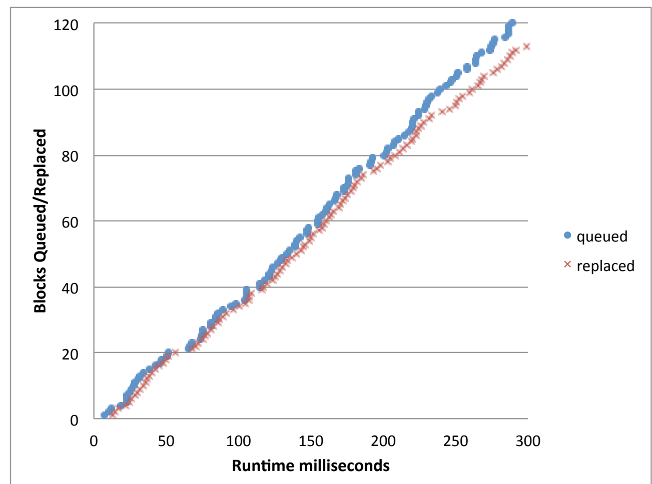


Figure 7: Zoom in on first part of figure 6.

The sheer number of blocks replaced in the graph above makes it difficult to discern the pattern of individual replacements. In order to see some of these figure 7 zooms in on the first part of of figure 6. As can be seen, even at this scale, the rate of queueing and replacement is relatively steady. However the small gaps visible in the graph indicate either:

1. A temporary lack of *need* to replace blocks due to the replacement queue being empty; or

2. A temporary lack of *opportunity* to replace blocks due to `cpu_exec()` finding all the blocks it needs in the map table, thus not triggering TCG/replacement; or

| Benchmark | raw size | opt size | reduction |
|---|---|---|---|
| hmmer | 79.7 | 44.8 | 41% |
| gcc | 85.44 | 46.6 | 42% |
| sjeng | 104.9 | 56.1 | 44% |

Table 3: Percentage of optimised blocks that are smaller and larger and the percentage of code saved on replaced blocks for three benchmarks

```
0x400818b4: add ip, pc, #0
0x400818b8: add ip, ip, #147456
0x400818bc: ldr pc, [ip, #1796]!
```

Figure 8: An example ARM code block

3. A temporary lack of *opportunity* to replace blocks because QEMU is running inside a chain in the translation cache and is not passing control to `cpu_exec()`.

The first scenario above is benign. The second and third are not. The second scenario can occur because we only trigger replacement when TCG is triggered. To test the impact of the second scenario we changed the replacement strategy so replacement could also occur at regular intervals when TCG is not triggered. We found that, across a range of benchmarks, this alternative strategy of periodic checking had almost no impact on the timing of replacements and a negative impact on runtimes (due to the maintenance of an interval counter in `cpu_exec()`). We tested for the third scenario and found that, while on most benchmarks QEMU spends only short intervals running chained blocks in the translation cache there were some benchmarks which spent long times. The mcf benchmark in particular spent a substantial proportion of its execution time running within a single chain. Such behaviour, while good for performance, has the potential to impact negatively on the current replacement strategy of augmented QEMU.

### 4.3 Code Size

Augmented QEMU produces significant reductions in the amount of code in the blocks it replaces. Table 3 shows results for three SPECint 2006 benchmarks, gcc, hmmer and sjeng on test workloads (adjusted for the inlined `cmp` instructions mentioned previously). The results show that almost all blocks that are optimised result in smaller code with substantial savings in space overall. In general we assume that this reduction in code size correlates with more efficient code. We look more closely at this issue next.

### 4.4 Code Quality

In this context we define code quality by efficiency and lack of redundancy. To ascertain the relative quality of the code being produced by the LLVM-JIT we performed a side-by-side inspection of a number of optimised and unoptimised cache blocks. Some of these revealed very substantial improvements. For example the ARM code block shown in figure 8 is translated by TCG into the reasonably compact x86 code shown in figure 9. However, the code produced by the LLVM-JIT is much better still, just two instructions:

```
mov %r14, %rdi
mov 0x400a58bc, 0x30(%rdi)
```

```
xor %r12d,%r12d
mov $0x400818bc,%r15d
add %r12d,%r15d
mov %r15d,0x30(%r14)
mov $0x24000,%r12d
mov 0x30(%r14),%r15d
add %r12d,%r15d
mov %r15d,0x30(%r14)
```

Figure 9: TCG translation of code from figure 8

the first of which is just moving the pointer to the CPU state struct. The savings come from eliminating a redundant move of the `pc` to `ip` and realising that, in this case `pc` can be recognised as a constant and folded in. These are just standard optimisations but TCG, which has to stamp its code out very quickly, doesnt have time to perform optimisations stretching over more than one ARM instruction.

The LLVM-JIT was also successful in removing a lot of redundant loads and stores which saved values out to the CPU state struct only to read them in again.

### 4.5 Code Speed

To test the raw speed of Augmented QEMU we ran 11 SPECint 2006 benchmarks on ARM code produced by GCC 4.4.3 (arm-softfloat-linux-gnueabi). All benchmarks were run on test workloads with the exception of specrand, which was run against its reference workload and gobmk which was run against its 13x13.txt reference workload in order to get longer runtimes.

Depending on length, each benchmark was run between 5 and 30 times and averaged. An exception was the gobmk benchmark which, due to its very long run-time on a reference workload was run twice. Absolute runtimes varied between three seconds for libquantum and 40 minutes for gobmk. In our experiments we measured real runtimes, with all QEMU logging turned off. The machine was dedicated to these experiments and so was lightly loaded at the time giving very consistent results - usually to within half-a-percent variation.

Figure 10 shows the relative speeds of:

**raw-qemu:** The time for vanilla-QEMU always normalised to one.

**no-replace:** The time for augmented-QEMU but without replacing any blocks – this gives an indication of overhead.

**replace:** The time for augmented-QEMU – blocks are replaced.

**net:** The net cost of augmented-QEMU assuming that there are no overheads.

Note, in order to make-visible the impact of overheads, the y-scale in figure 10 starts at 0.75. Speedup between vanilla-QEMU and augmented-QEMU varied from negative one percent on gcc, omnetpp, and specrand to 12 percent on the mcf benchmark. Speedup seemed not to correlate strongly with the size of the benchmark in terms of run time or in terms of the number of blocks it contained. It can be conjectured that the benchmarks that exhibited some regularity in their computation over time gained most but this will require further investigation to confirm.

The overhead of running augmented-QEMU can be estimated as the difference between raw-qemu and

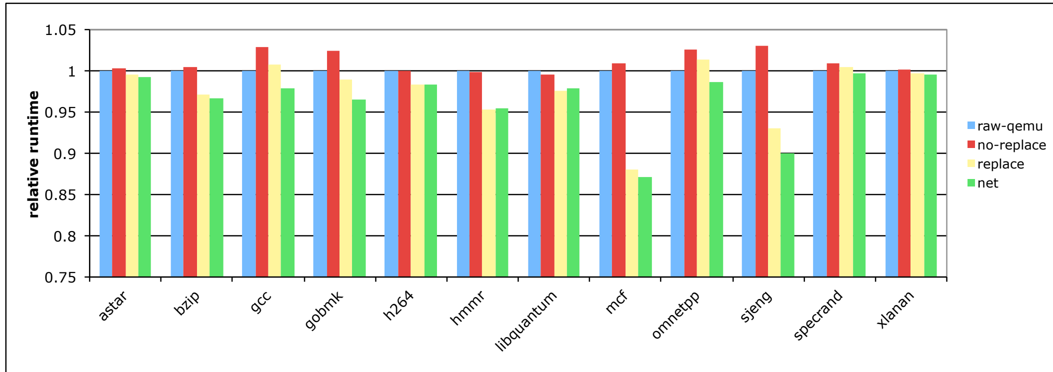


Figure 10: Relative run-times of various forms of QEMU on SPECint 2006 benchmarks. (note: scale starts at 0.75).

no-replace in figure 10. In our experiments with the sjeng benchmarks almost all of this overhead is the cost of detecting which blocks are eligible for translation. Running the translation and the LLVM-JIT in a separate thread had no discernible impact on times and locks on the ring-buffers suffered almost no contention.

If the costs of the overhead are subtracted from the augmented-QEMU run-times then we have a rough estimate of the impact of code improvement sans overheads. The last *net* column for each benchmark in figure 10 represents this measure. In all cases the impact of the code replacement was either positive or neutral[7].

Note that any of the improvements shown above can only come from TBs that are run after replacement. If a replaced TB is run only for a short time or not run at all after replacement it can have very limited impact. We briefly investigated this issue by modifying the test workload input to the sjeng benchmark so it was forced to repeat the computation involved in the test workload three times. This gives more time for the replaced blocks to have an impact in reducing runtime. We found that there was very little reduction in runtime from running with the replaced blocks for longer. This seems to indicate, that on this benchmark, that the replacement-blocks that reduce runtime appear in the translation-cache early.

## 5 Conclusions, Limitations and Future work

In this article we have described Augmented-QEMU, a set of small modifications to QEMU that leverage the LLVM-JIT, running on a separate processor core, to improve simulation performance. This work demonstrates that it is feasible to improve the speed of an already-fast ISS by conservative, incremental, and minimally intrusive changes using an established compiler framework. While work to date is promising, it is a work in progress, and there there are a number of limitations to the current framework that we plan to address in our future work. These limitations are:

**Code Coverage** Currently, not all ARM instructions have translations to LLVM-IR. We will continue to implement these incrementally to improve coverage and performance.

**Single Background Core** Currently we only use one core for optimisation. This limitation is partly due to LLVM versions 2.6 and, to a lesser extent, 2.7 not being perfectly amenable to multi-threaded execution. This limitation appears to be addressed in version 2.9 and we plan to expand background optimisation to more cores using this version.

**ARM-only Guest** The current implementation is specialised to translate only ARM guest code. We made this choice primarily because of the expressiveness and relative ease of translation in the LLVM framework of the ARM instruction set relative to TCG intermediate code. However, full-portability, could be achieved if we do translate from TCG intermediate code to LLVM IR and this is a worthy future goal.

**Timing of Replacement** The limiting of block replacement to when TCG is running is safe and is low-overhead but can potentially lead to blocks not being replaced until after they are needed. A better future strategy is to build our own translation cache, complete with chaining, and perform thread-safe fix-ups to do indirect jumps to the translation cache.

**Exception Emulation** QEMU keeps track of every register in the processor state except the SPC while it runs the the TB. When an exception or interrupt is triggered vanilla-QEMU rebuilds the real value of the SPC by abstractly re-executing from the start of the current TB, keeping track of what would happen to the SPC on the way. When the host PC reaches the point at which the exception was triggered the SPC is recorded in the processor state and the exception can then be handled. Our block replacement strategy prevents this abstract-re-execution to get the SPC value. While this is not an issue for SPECint benchmarks in user-mode QEMU it will need to be addressed in future. One good way to address this is to build our own translation cache, this would leave QEMU's Translation Cache intact allowing QEMU's current mechanism to work.

Note that the last two of these limitations can be addressed by building a separate translation-cache for our LLVM-JIT generated code. A new code cache would also open up opportunities to improve chaining form and perform new inter-block optimisations. Finally, using a separate code cache would give us opportunity to build super-blocks with the potential to be efficiently saved either in binary form or as LLVM-IR to speed up future runs as is done in SIMIT-ARM. We believe that an incrementally constructed

[7]Though the libquantum benchmark actually displayed a very slighly shorter runtime in the no-replace case. This is likely to be due to sampling noise between runs since libquantum has a very short run-time.

combination of background optimisation, building of large chains, and code persistence has the potential to greatly increase the execution speed of ISSs in future.

## References


K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS*, pages 2–13, 2006.

O. Almer, I. Böhm, T. E. von Koch, B. Franke, S. Kyle, V. Seeker, C. Thompson, and N. Topham. Scalable multi-core simulation using parallel dynamic binary translation. In *International Symposium on Systems, Architectures, Modelling and Simulation (SAMOS'11)*, 2011.

B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The jikes research virtual machine project: Building an open-source research community. *IBM Systems Journal*, 44(2):399 –417, 2005. ISSN 0018-8670. doi: 10.1147/sj.442.0399.

M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449 –466, 2005. ISSN 0018-9219. doi: 10.1109/JPROC.2004.840305.

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

D. Bruening, T. Garnett, and S. P. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO*, pages 265–275, 2003.

A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. Bharadwaj Yadavalli, and J. Yates. Fx!32 a profile-directed binary translator. *Micro, IEEE*, 18(2):56 –64, 1998. ISSN 0272-1732. doi: 10.1109/40.671403.

V. Chipounov and G. Candea. Dynamically Translating x86 to LLVM using QEMU. Technical report, 2010.

R. F. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *SIGMETRICS*, pages 128–137, 1994.

G. De Michell and R. Gupta. Hardware/software co-design. *Proceedings of the IEEE*, 85(3):349 –365, Mar. 1997. ISSN 0018-9219. doi: 10.1109/5.558708.

Google .Inc. Android emulator, 2011.

A. Jeffery. Using the llvm compiler infrastructure for optimised, asynchronous dynamic translation in qemu. Master's thesis, School of Computer Science, University of Adelaide, 2010.

W. Jiang, Y. Zhou, Y. Cui, W. Feng, Y. Chen, Y. Shi, and Q. Wu. Cfs optimizations to kvm threads on multi-core environment. In *ICPADS*, pages 348–354, 2009.

C. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *Softw., Pract. Exper.*, 31(8):717–738, 2001.

P. A. Kulkarni, M. Arnold, and M. Hind. Dynamic compilation: the benefits of early investing. In *VEE*, pages 94–104, 2007.

C. Lattner and V. S. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.

J. Lee, J. Kim, C. Jang, S. Kim, B. Egger, K. Kim, and S. Han. Facsim: a fast and cycle-accurate architecture simulator for embedded systems. In *LCTES*, pages 89–100, 2008.

W. H. Lee. Arm instruction set simulation on multi-core x86 hardware. Master's thesis, School of Computer Science, University of Adelaide, 2009.

LLVM-Project. Llvm users, 2011. URL `http://llvm.org/Users.html`.

P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50 –58, Feb. 2002. ISSN 0018-9162. doi: 10.1109/2.982916.

OVP, April 2011. URL `http://www.ovpworld.org/`.

W. Qin, J. D'Errico, and X. Zhu. A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation. In *CODES+ISSS*, pages 193–198, 2006.

T. Scheller. LLVM-QEMU, Google Summer of Code Project, 2008.

Standard-Performance-Evaluation-Corporation. Spec cpu2006 cint2006 benchmarks, 2011. URL `http://www.spec.org/cpu2006/`.

N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. Rifle: An architectural framework for user-centric information-flow security. In *MICRO*, pages 243–254, 2004.

J. Watson. Virtualbox: bits and bytes masquerading as machines. *Linux J.*, 2008, February 2008. ISSN 1075-3583.

D. Wentzlaff and A. Agarwal. Constructing virtual architectures on a tiled processor. In *CGO*, pages 173–184, 2006.

E. Witchel and M. Rosenblum. Embra: fast and flexible machine simulation. *SIGMETRICS Perform. Eval. Rev.*, 24:68–79, May 1996. ISSN 0163-5999. doi: http://doi.acm.org/10.1145/233008.233025. URL `http://doi.acm.org/10.1145/233008.233025`.


We'd like to thank Tillmann Scheller for sharing his insights from the LLVM-QEMU project; Wanghao Lee for invaluable early input on this project; and the research team at ASTC for their expert advice and enthusiastic support.