ABSTRACT
Modern processor design tools integrate in their workflows generators for instruction set simulators (Iss) from architecture descriptions. Whilst these generated simulators are useful for design evaluation and software development, they suffer from poor performance. We present an ultra-fast Jit-compiled Iss generated from an ArchC description. We also introduce a novel partial evaluation optimisation, which further improves Jit compilation time and code quality. This results in a simulation rate of 510MIPS for an Arm target across 45 EEMBC and Spec benchmarks. On average, our Iss is 1.7 times faster than Simit-Arm, one of the fastest Iss generated from an architecture description.

1. INTRODUCTION
Instruction set simulators (Iss) are indispensable tools for hardware architects and software developers alike. Consequently, modern processor design suites such as Synopsys Processor Designer [13] or Target IP Designer [14] integrate Iss generator tools, which synthesise Iss from high-level architecture descriptions. However, these machine generated simulators are typically slower than their hand-coded counterparts targeting only a single architecture for which they have been optimised [10].

Currently, the fastest available Iss are based on dynamic binary translation (Dbt) and use a parallel just-in-time (Jit) compiler [4, 11] for the translation of regions of target machine code to the host system’s Isa, possibly interleaving a detailed performance model of the simulated processor for cycle-accurate pipeline modelling [5]. Unfortunately, these fast Dbt based Iss are – for performance reasons – hand-coded and, thus not easily retargetable. Naïve approaches to generating Jit Dbt simulators from high-level architectural descriptions suffer from two problems: poor quality of the Jit compiled code and excessive Jit compilation times. The common root of these two problems lies in the complex behaviours of target machine instructions containing different possible execution paths dependent on, for example, processor state.

In this paper we present a novel approach to generating ultra-fast Jit Dbt simulators from high-level ArchC based architecture descriptions [1]. We apply a partial evaluation optimisation [8], which eliminates dead execution paths from complex instruction behaviours, early on in the Jit compilation process as part of the intermediate representation (Ir) generation stage. This not only relieves the underlying low-level Jit compiler from performing this work, which possibly utilises expensive analyses, but aids better code generation. In turn, this results in faster Jit compilation and, at the same time, less – and less complex – code is emitted by the Jit compiler due to compile-time specialisation.

The key idea is to simplify instruction behaviours as soon as possible in the high-level Jit code generator (see Figure 2), rather than deferring optimisation of the generated code to the low-level Jit compiler (see Figure 1). Through the use of early partial evaluation at Ir generation time we ensure that instruction behaviours that are impossible in a specific context are not emitted, and thus are never presented to the low-level Jit compiler. Rather than emitting complex instruction behaviour patterns and relying on the low-level Jit compiler to eliminate dead code later, we spend a little more effort in the high-level Ir generator. In return, we gain significantly higher performance in, and achieve better code quality from, the low-level Jit compiler, resulting in greater overall simulation speed.

We have evaluated our Iss generation approach using an Arm v5 architecture model written in ArchC against 45 EEMBC and Spec Cpu2006 benchmarks. On a standard x86 simulation host we demonstrate an average simulation rate of 191 MIPS for our baseline Iss using a naïve generation scheme (see Figure 1), and 510 MIPS after enabling early partial evaluation (see Figure 2). Using the same Arm v5 target and the same simulation host machine (see Table 1) this is approximately 21 times faster than the original ArchC simulator (24 MIPS) [1], 78 times faster than FaC-Sim (6.5 MIPS) [9], 1.7 times faster than Simt-Arm (300 MIPS) [11], which relies heavily on manual instruction specialisation as part of the modelling and retargeting process, and only 24.5% slower than Qemu-Arm [2], which has been hand-tuned for the target, requires significant low-level retargeting effort and unlike our Iss sacrifices instruction observability for performance.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC ’13, May 29 – June 07 2013, Austin, TX, USA.
Copyright 2013 ACM 978-1-4503-2071-9/13/05 ...$15.00.
In contrast, our early partial evaluation scheme already evaluates statically evaluable expressions (lines 1, 3 and 4) at IR generation time and avoids emitting dead code such as the flag setting code in lines 7–13 of Listing 2. This means, we produce less IR for the code generator to process and, hence, we speed up the JIT compiler.

For the example in Figure 1 our partial evaluation scheme is able to fully compute the result of the example add instruction at compile-time (see Listing 3). In comparison, the naïve scheme requires much more expensive 4-03 LLVM optimisations to achieve a similar effect. The naïve scheme generates more LLVM IR instructions, which are then processed by the optimiser, resulting in a long latency of the performance-critical JIT compiler.

2. METHODOLOGY

2.1 Simulator Generation

Our simulator generation system is based on there being a distinct boundary between the processor implementation and the simulator framework (see Figure 4). Processor models are written using a variant of ArchC [1], and then processed into a set of C++ source files implementing instruction decoding, an interpreter, a JIT module, disassembly, etc. This system is sufficiently general and flexible to support a variety of ISAs (so far we have models for ARM V5, ARM V7A, ARM V7M, PowerPC, Intel 8086 and MOS 6502) and is modular enough that different implementations of interpreters, JIT systems, etc can be swapped in and out.

The baseline JIT system precompiles each instruction implementation to LLVM bitcode functions. When an instruction is JIT compiled, a call to the appropriate function is generated (with the instruction fields as parameters) and inlined.

2.2 Early Partial Evaluation

The early partial evaluation JIT is generated completely from a processor ISA description. The implementation of each instruction is written in a C-like language, and compiled into an SSA form consisting of variable reads and writes, register and memory reads and writes, unary and binary operations, some intrinsics, and control flow. The SSA form does not have Φ-nodes, since Φ-analysis is unnecessary at this stage and can be done later by the LLVM back-end. Function calls and subroutines are supported, and all func-
Algorithm 1 Computing ‘fixedness’ of variable modifying statements in an instruction implementation.

```plaintext
function INSSIMPLEFIXEDNESS(action)
  for all b ∈ BB ∈ action do
    b.dyn_in ← []
    b.dyn_out ← []
    b.ctrlflow ← invalid
    b.mark_variable_accesses_as_fixed()
    wl ← [action.entry_block]
  while wl is not empty do
    b ← wl.pop_front()
    result ← BBFIXEDNESS(b)
    if result = False then
      wl.insert(b.successors)
  end

function BBFIXEDNESS(block)
  for all p ∈ BB ∈ block.predecessors do
    if p.ctrlflow = dynamic ∧ p.final_stmt.is_fixed then
      block.ctrlflow ← dynamic
      block.dyn_in ← block.dyn_in ∪ p.dyn_out
      dyn_now ← block.dyn_in
      for all s ∈ Statement ∈ block.statements do
        if s writes a dynamic value to a variable v then
          dynamic_now ← v
        if s reads a variable in dyn_now then
          mark s as dynamic
          block.dyn_out ← dyn_now
          if block.ctrlflow changed || dyn_now ≠ dyn_in then
            return False
        end
      end
    end
  end
  return True
```

The partial evaluation scheme itself is based on computing whether or not each SSA statement is ‘fixed’ (i.e., relies only on static-time information such as instruction fields and constant values) or ‘dynamic’ (i.e., relies on information loaded from registers or memory). For most statements, this is trivial: a binary operation is fixed if both of its operands are fixed, a memory or register write is fixed if its address and value are fixed etc. However, computing the fixedness of variable reads (and thus statements which depend on those variable reads) requires slightly more careful analysis.

The algorithm for doing this analysis can be seen in Algorithm 1. The analysis is done on each instruction implementation. Each basic block in each implementation has three variables: a list of variables which are ‘dynamic’ at the entry point of the block, a list of variables dynamic at the exit of the block, and a variable stating whether or not control flow into the block is fixed or dynamic. We start by emptying the lists and considering the control flow ‘invalid’ (not yet computed).

We then process each block in turn, starting with the entry block of the instruction. If the block processing algorithm reports that the state of the block has changed (either its control flow or dynamic IN/OUT lists) then we add its successors to the work list. Once the work list becomes empty the fixedness computation is complete. A single block may be processed multiple times if it is in the body of a loop.

For each basic block we first compute whether or not control flow into the block is fixed or dynamic. Control flow is dynamic if any conditional branch or switch in any ancestor of the block depends on a dynamic value. Otherwise, control flow into the block is fixed. We also compute the dynamic IN variables as the union of the dynamic OUT variables of the block’s predecessors.

We then loop over each of the statements in the block in program order. For each statement, if the statement writes...
dynamic values to any variables, we add those variables to the dynamic OUT list. For variable reads, if the read variable is dynamic at this point, we mark the read as dynamic.

### 2.3 JIT Compiler Generation

When generating the JIT compiler itself, we again work on an instruction type by instruction type basis. For each instruction type we generate a function which implements the fixed portions of the instruction directly, and generates LLVM instructions for the dynamic portions. So, fixed control and data flow in the instruction becomes C++ control and data flow, and dynamic control and data flow becomes LLVM control and data flow.

Since a variable may be fixed at one point during execution but dynamic later on, we do further analysis to determine points at which variables must be ‘spilled’ into the dynamic context. This is only strictly necessary when a variable is sometimes dynamic on entry to a block (e.g. if we write a dynamic value to it in the ‘then’ portion of an ‘if’ statement but write a static value in the ‘else’ portion). If a dynamic statement reads the value of a fixed variable we write the value directly into the output LLVM statement rather than ‘spilling’ the variable.

As static control flow is executed completely at JIT time, multiple SSA blocks may become a single LLVM block. This produces one of the main improvements in code generation speed in the form of ‘on-the-fly’ dead code elimination, meaning that there is much less code for LLVM to generate and optimise. Blocks which have dynamic control flow are only emitted ‘on-demand’ (i.e., if either a static or dynamic control flow statement which has the block as a target is encountered) which also helps to reduce the amount of code passed into LLVM.

### 3. EMPIRICAL EVALUATION

We have implemented the presented early partial evaluation technique in our Iss framework and evaluated it against the full EEMBC 1.1 and SPEC CINT2006 benchmark suites. We have compiled the benchmarks with the ARM v5 port of the GCC 4.5.2 compiler. For each of the benchmarks we have measured the time from start of the simulator to completion, with and without early partial evaluation, and used these times to calculate absolute simulation rates (in MIPS) and speedups. For comparison, we have executed the same binaries of the benchmarks on SimT-ARM v3 [11] Iss using an equivalent configuration and performed the same timing measurements. Details of the simulation host, the target platform and Iss configuration are summarised in Table 1.

### 3.1 Key Performance Results

Our main results showing relative speedups resulting from our early partial evaluation scheme over the parallel, JIT-compiled SimT-ARM v3 Iss for both short-running EEMBC and long-running SPEC Cpu2006 benchmarks are presented in Figure 5.

For all but two of the EEMBC benchmarks our partial evaluation Iss significantly outperforms SimT-ARM v3. On individual benchmarks (cacheb01 and xdttrn01) speedups of up to nearly 4 can be observed, with a geometric mean of 1.86 over all EEMBC benchmarks. This indicates that for these short-running benchmarks, where JIT compilation times contribute for a relatively larger amount of the overall execution time, partial evaluation has a large positive effect.

Although the performance of our JIT system is very good, there are a few benchmarks in the EEMBC suite in which it does not perform as well as SimT-ARM, namely the rotate01 and bitmap01 benchmarks. On closer examination, these benchmarks are extremely heavy in control flow. The rotate01 benchmark, on which SimT-ARM performed the

---

**Table 1: Simulation host, target platform and Iss.**

<table>
<thead>
<tr>
<th>Simulation Host</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon X5650</td>
<td></td>
</tr>
<tr>
<td>Frequency</td>
<td>2.67GHz</td>
<td></td>
</tr>
<tr>
<td># Cores</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Target Platform</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ISA</td>
<td>ARM v5</td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td>GCC 4.5.2</td>
<td></td>
</tr>
<tr>
<td>Executable Format</td>
<td>Linux ELF</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Simulator Configuration</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td># JIT threads</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>JIT compiler</td>
<td>LLVM 2.9</td>
<td></td>
</tr>
<tr>
<td>JIT optimisation flags</td>
<td>-O3</td>
<td></td>
</tr>
<tr>
<td>System calls</td>
<td>Emulated</td>
<td></td>
</tr>
</tbody>
</table>
3.2 Code Size Analysis

One of the main benefits of using our partial evaluation scheme is the extreme code size reduction benefit. Figure 6 shows the reduction in LLVM bitcode emitted at standard ‘-O1’ and ‘-O3’ optimisation levels when using our scheme compared to a naive scheme. An 80% reduction in code size is obtained when optimising to ‘-O1’ (i.e., the partial evaluation scheme emits only 20% as many instructions as the naive scheme) on the EEMBC suite, and an even larger reduction is seen for SPEC. A smaller but still significant reduction in code size is observed when optimising to ‘-O3’. This produces a significant reduction in compilation time which can be exploited to improve JIT warmup speed (i.e., translate code faster) or to more aggressively optimise translations.
4. RELATED WORK

Retargetable Iss have seen an increased interest in recent years. SimT-Arm [11] is the Arm V5 port of such an Iss, where retargeting is achieved through provision of a high-level isa model from which C language simulator modules are generated. SimT-Arm uses parallel and distributed Jit compilation for dynamic code generation, which makes it one of the fastest available Iss today. Still, instruction specialisation is left to the user to be performed manually as part of the retargeting and modelling process. In [7] a simulation framework is presented, which is based on a structural architecture description language (Adl) and that uses the LLVM open-source compiler infrastructure to dynamically translate instruction sequences of the simulated architecture into machine instructions of the host machine. The code generation scheme in [7] is relatively simple and operates at basic block level and relies entirely on LLVM standard optimisations. In contrast, our simulator operates on trace regions and adds partial evaluation on top of standard code optimisations, thus providing higher performance without manual user intervention. Another popular, retargetable Iss is QEMU [2]. Retargeting of QEMU involves rewriting blocks of target code using low-level tiny code generator (TcG) operations, which are a machine-independent intermediate notation. Subsequently this notation is being compiled for the host’s architecture by TcG subject to optional optimisation passes.Unlike our simulator and also SimT-Arm, which employ high-level isa descriptions, TcG requires that there be dedicated low-level TcG code written to support each target instruction.

An interesting approach is presented in [3]. It aims at generating an Iss from a pseudo-formal document such as a datasheet. However, this approach still requires lots of manual adaptation. In [12] specialisation of instruction behaviours in a generated Iss is discussed.

The open-source ArchC tool-suite [1] contains an Iss generator, which is based on the same architecture description language as we are using. The ArchC simulation methodology, however, is very different in that they translate the architecture model to a set of systemC classes for processor modelling, whereas we generate a Jit based Iss. As a result our simulator is more than one order of magnitude faster than the original ArchC simulator.

Commercially available, retargetable Iss are included in the Synopsys Processor Designer [13] and Target Ip Designer [14] tool suites. These packages are aimed at Asp design support and unlike the simulator presented here offer less support for complex features typically found in general-purpose embedded processors. We explicitly consider user and kernel mode operation, interrupt handling, memory management and make provisions for efficiently handling self-modifying code.

Partial evaluation [8] is a widely-used program optimisation based on specialisation, which precomputes effects of static input at compile time, thus reducing the complexity and runtime of the generated code. In Jit compilation partial evaluation has the potential to both reduce compilation time and increase code quality. This has been demonstrated, for example, in the PyPy Python Jit compiler [6].

5. SUMMARY AND CONCLUSION

In this paper we have presented an early code optimisation based on partial evaluation, which can be applied in Jit-compiled Iss generated from high-level architecture descriptions. We have demonstrated by implementation that early partial evaluation reduces the overhead of Jit compilation and improves the quality of the generated code, hence contributing to increased overall performance of the Iss. For an Arm V5 architecture model evaluated against the Eembc and Spec Cpu2006 benchmarks our Iss delivers an average simulation rate of 510 MIPS, outperforming the state-of-the-art, Adl-retargetable SimT-Arm Iss by as much as 297%.

6. REFERENCES