Efficient Dual-ISA Support in a Retargetable, Asynchronous Dynamic Binary Translator

Citation for published version:

Digital Object Identifier (DOI):
10.1109/SAMOS.2015.7363665

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Efficient Dual-ISA Support in a Retargetable, Asynchronous Dynamic Binary Translator

Tom Spink, Harry Wagstaff, Björn Franke and Nigel Topham
Institute for Computing Systems Architecture
School of Informatics, University of Edinburgh
t.spink@sms.ed.ac.uk, h.wagstaff@sms.ed.ac.uk, bfranke@inf.ed.ac.uk, npt@inf.ed.ac.uk

Abstract—Dynamic Binary Translation (DBT) allows software compiled for one Instruction Set Architecture (ISA) to be executed on a processor supporting a different ISA. Some modern DBT systems decouple their main execution loop from the built-in Just-In-Time (JIT) compiler, i.e. the JIT compiler can operate asynchronously in a different thread without blocking program execution. However, this creates a problem for target architectures with dual-ISA support such as ARM/THUMB, where the ISA of the currently executed instruction stream may be different to the one processed by the JIT compiler due to their decoupled operation and dynamic mode changes. In this paper we present a new approach for dual-ISA support in such an asynchronous DBT system, which integrates ISA mode tracking and hot-swapping of software instruction decoders. We demonstrate how this can be achieved in a retargetable DBT system, where the target ISA is not hard-coded, but a processor-specific module is generated from a high-level architecture description. We have implemented ARM V5T support in our DBT and demonstrate execution rates of up to 1148 MIPS for the SPEC CPU 2006 benchmarks compiled for ARM/THUMB, achieving on average 192%, and up to 323%, of the speed of QEMU, which has been subject to intensive manual performance tuning and requires significant low-level effort for retargeting.

I. INTRODUCTION

The provision of a compact 16-bit instruction set architecture (ISA) alongside a standard, full-width 32-bit RISC ISA is a popular architectural approach to code size reduction. For example, some ARM processors (e.g. ARM7TDMI) implement the compact THUMB instruction set whereas MIPS has a similar offering called MIPS16E. Common to these compact 16-bit ISAs is that the processor either operates in 16-bit or 32-bit mode and switching between modes of operation is done explicitly through mode change instructions, or implicitly through PC load instructions.

For instruction set simulators (ISS), especially those using dynamic binary translation (DBT) technology rather than instruction-by-instruction interpretation only, dynamic changes of the ISA present a challenge. Their integrated instruction decoder, part of the just-in-time (JIT) compiler translating from the target to the host system’s ISA, needs to support two different instruction encodings and keep track of the current mode of operation. This is a particularly difficult problem if the JIT compiler is decoupled from the main execution loop and, for performance reasons, operates asynchronously in a different thread as in e.g. [1] or [2]. For such asynchronously multi-threaded DBT systems, the ISA of the currently executed fragment of code may be different to the one currently processed by the JIT compiler. In fact, in the presence of a JIT compilation task farm [2], each JIT compilation worker may independently change its target ISA based on the encoding of the code fragment it is operating on. Most DBT systems [3], [4], [5], [6], [7], [8], [9], [1], [10], [11], [12], [13], [14], [15] avoid dealing with this added complexity and do not provide support for dual-ISAs at all. A notable exception is the ARM port of QEMU [16], which supports both ARM and THUMB instructions, but tightly couples its JIT compiler and main execution loop and, thus, misses the opportunity to offload the JIT compiler from the critical path to a separate thread.

The added complexity and possible performance implications of handling dual ISAs in DBT systems motivate us to investigate high-level retargetability, where low-level implementation and code generation details are hidden from the user. In our system ISA modes, instruction formats and behaviours are specified using a C-based architecture description language (ADL), which is processed by a generator tool that creates a dynamically loadable processor module. This processor module encapsulates the necessary ISA tracking logic, instruction decoder trees and target instruction implementations. Users of our system can entirely focus on the task of transcribing instruction definitions from the processor manual and are relieved of the burden of writing or modifying DBT-internal code concerning ISA mode switches.

In this paper we introduce a set of novel techniques enabling dual-ISA support in asynchronous DBT systems, involving ISA mode tracking and hot-swapping of software instruction decoders. The key ideas can be summarised as follows: First, for ISA mode tracking we annotate regions of code discovered during initial interpretive execution with their target ISA. This information cannot be determined purely statically. Second, we maintain separate instruction decoder trees for both ISAs and dynamically switch between software instruction decoders in the JIT compiler according to the annotation on the code region under translation. Maintaining two instruction decoder trees, one for each ISA, contributes to efficiency. The alternative solution, a combined decoder tree, would require repeated mode checks to be performed as opcodes and fields of both ISAs may overlap. Finally, we demonstrate how dual-ISA support can be integrated in a retargetable DBT system, where both the interpreter and JIT compiler, including their instruction decoders and code generators, are generated from a high-level architecture description. We have implemented full ARM V5T support, including complete coverage of THUMB instructions, in our retargetable, asynchronous DBT system and evaluated it against the SPEC CPU 2006 benchmark suite. Using the ARM port of the GCC compiler we have compiled the
benchmarks for dual-ISA ARM and THUMB execution. Across all benchmarks we achieve an average execution rate of 780.56 MIPS, which is 28% faster than the single-ISA performance, demonstrating the high efficiency of our approach. Leveraging our asynchronous JIT compiler our automatically generated DBT system archives on average 192% of the performance of QEMU-ARM, which has been manually optimised using detailed knowledge of its low-level TCG code generator.

A. Translation/Execution Models for DBT Systems

Before describing our contributions we review existing translation/execution models for DBT systems with respect to their ability to support target dual instruction sets.

1) Single-mode translation/execution model

a) Interpreter only. In this mode the entire target program is executed on an instruction-by-instruction basis. Strictly, this is not DBT as no translation takes place. It is straight-forward to keep track of the current ISA as mode changes operations take immediate effect and the interpreter can handle the next instruction appropriately based on its current state (see Figure 1(a)). ISS using interpretative execution such as SIMPLESCALAR [9] or ARMISS [15] have low implementation complexity, but suffer from poor performance.

b) JIT only. Interpreter-less DBT systems exclusively rely on JIT compilation to translate every target instruction to native code before executing this code. As a consequence, execution in this model will pause as soon as previously unseen code has been discovered and only resume after JIT compilation has completed. ISA mode changes take immediate effect (see Figure 1(b)) and are again simple to implement as native code execution and JIT compilation stages are tightly coupled and mutually exclusive. JIT-only DBT systems are of low complexity and provide better performance than purely interpreted ones, but rely on very fast JIT compilers, which in turn will often perform very little code optimisation. This and the fact that the JIT compiler is on the critical path of the main execution loop within a single thread limits the achievable performance. QEMU [16], STRATA [6], [7], [8], SHADE [4], SPIRE [17], and PIN [18] are based on this model.

2) Mixed-mode translation/execution model

a) Synchronous (single-threaded). This model combines both an interpreter and a JIT compiler in a single DBT (see Figure 1(c)). Initially, the interpreter is used for code execution and profiling. Once a region of hot code has been discovered, the JIT compiler is employed to translate this region to faster native code. The advantage of a mixed-mode translation/execution model is that only profitable program regions are JIT translated, whereas infrequently executed code can be handled in the interpreter without incurring JIT compilation overheads [19]. Due to its synchronous nature ISA tracking is simple in this model: the current machine state is available in the interpreter and can be used to select the appropriate instruction decoder in the JIT compiler. As before, the JIT compiler operates in the same thread as the main execution loop and program execution pauses whilst code is translated. This limits overall performance, especially during prolonged code translation phases. A popular representative of this model is DYNAMO [20].

b) Asynchronous (multi-threaded). This model is characterised by its multi-threaded operation of the main execution loop and JIT compiler. Similar to the synchronous mixed-mode case, an interpreter is used for initial code execution and discovery of hotspots. However, in this model the interpreter enqueues hot code regions to be translated by the JIT compiler and continues operation without blocking (see Figure 1(d)). As soon as the JIT compiler installs the native code the execution mode switches over from interpreted to native code execution. Only in this model is it possible to leverage concurrent JIT compilation on multi-core host machines, hiding the latency of JIT compilation and, ultimately, contributing to higher performance of the DBT system [1]. [2]. Unfortunately, this model presents a challenge to implementing dual-ISA support: the current machine state represented in the interpreter may have advanced due to its concurrent operation and cannot be used to appropriately decode target instructions in the JIT compiler.

In summary, decoupling the JIT compiler from main execution loop and offloading it to a separate thread has been demonstrated to increase performance in multi-threaded DBT systems. However, it remains an unsolved problem how to efficiently handle dynamic changes of the target ISA without tightly coupling the JIT compiler and, thus, losing the benefits of its asynchronous operation.

B. Motivating Example

The nature of the ARM and THUMB instruction set is such that it is not possible to statically determine from the binary encoding alone which ISA the instruction is part of. This becomes even more important when it is noted that ARM instructions are 32-bit in length, and THUMB instructions are
16-bit. For example, consider the 32-bit word e2810006. An ARM instruction decoder would decode the instruction as:

\[
\text{add } r0, r1, \#6
\]

whereas, a THUMB instruction decoder would consider the above 32-bit word as two 16-bit words, and would decode as the following two THUMB instructions:

\[
\text{mov } r6, r0 \\
\text{b.n } +4
\]

An ARM processor correctly decodes the instruction by being in one of two dynamic modes: ARM or THUMB.

A disassembler, given a sequence of instructions, has no information about what ISA the instructions belong to, and can therefore not make the distinction between ARM and THUMB instructions on a raw instruction stream, and must use debugging information provided with the binary to perform disassembly. If the debugging information is not available (e.g. it has been “striped” from the binary) then the disassembler must be instructed how to decode the instructions (assuming the programmer knows), and if the instructions are mixed-mode, then it will not be able to effectively decode at all. This problem for disassemblers directly translates to the same problem in any DBT with multi-ISA support. A DBT necessarily works on a raw instruction stream – without debugging information – and therefore must use its own mechanisms to correctly decode instructions. In the example of an ARM/THUMB DBT, it may choose to simulate a THUMB status bit as part of the CPSR register existent in the ARM architecture (see Section II), and therefore use the information within the register to determine how the current instruction should be decoded. But as mentioned in Section I-A, this approach does not work in the context of an asynchronous JIT compiler, as the state of the CPSR within the interpreter would be out of sync with the compiler during code translation.

C. Overview

The remainder of this paper is structured as follows. We review the dual ARM/THUMB ISA as far as relevant for this paper in Section II. We then introduce our new methodology for dual-ISA DBT support in Section III. This is followed by the presentation of our experimental evaluation in Section IV and a discussion of related work in Section V. Finally, in Section VI we summarise our findings and conclude.

II. BACKGROUND: ARM/THUMB

THUMB is a compact 16-bit instruction set supported by many ARM cores in addition to their standard 32-bit ARM ISA. Internally, narrow THUMB instructions are decoded to standard ARM instructions, i.e. each THUMB instruction has a 32-bit counterpart, but the inverse is not true. In THUMB mode only 8 out of the 16 32-bit general-purpose ARM registers are accessible, whereas in ARM mode no such restrictions apply. The narrower 16-bit instructions offer memory advantages such as increased code density and higher performance for systems with slow memory. The Current Program Status Register (CPSR) holds the processor mode (user or exception flag), interrupt mask bits, condition codes, and THUMB status bit. The THUMB status bit (T) indicates the processor’s current state: 0 for ARM state (default) or 1 for THUMB. A saved copy of CPSR, which is called Saved Program Status Register (SPSR), is for exception mode only. The usual method to enter or leave the THUMB state is via the Branch and Exchange (BX) or Branch, Link, and Exchange (BLX) instructions, but nearly every instruction that is permitted to update the PC may make a mode transition. During the branch, the CPU examines the least significant bit (LSB) of the destination address to determine the new state. Since all ARM instructions are aligned on either a 32- or 16-bit boundary, the LSB of the address is not used in the branch directly. However, if the LSB is 1 when branching from ARM state, the processor switches to THUMB state before it begins executing from the new address; if 0 when branching from THUMB state, the processor changes back to ARM state. The LSB is also set (or cleared) in the LR to support returning from functions that were called from a different mode. When an exception occurs, the processor automatically begins executing in ARM state at the address of the exception vector, even if the CPU is running in THUMB state when that exception occurs. When returning from the processor’s exception mode, the saved value of T in the SPSR register is used to restore the state. This bit can be used, for example, by an operating system to manually restart a task in the THUMB state – if that is how it was running previously.

III. METHODOLOGY: DUAL-ISA DBT SUPPORT

The DBT consists of an execution engine and a compilation engine. The execution engine will execute either native code (which has been generated from instructions by the compilation engine) or will execute instructions in an interpreter loop. The execution engine interpreter will also generate profiling data to pass to the compilation engine (see Figure 2). The execution engine maintains a machine state structure, within
which is contained the current execution mode of the target processor (along with other state information, such as register values etc). The machine state is only available to the execution engine, as the asynchronous compilation engine does not run in sync with the currently executing code. The compilation engine accepts compilation work units generated by the profiling component of the interpreter. A compilation work unit contains a control-flow graph (fundamentally a list of basic-blocks and their associated successor blocks) that are to be compiled. Each basic-block also contains the ISA mode that the instructions within the block should be decoded with.

### A. ISA Mode Tracking

The current ISA mode of the CPU is stored in a CPU state variable, which is updated in sequence as the instructions of the program are being executed. When the interpreter needs to decode an instruction (and cannot retrieve the decoding from the decoder cache), the current mode is looked up from the state variable and sent to the decoder service, which then decodes the instruction using the correct ISA decode tree. If an instruction causes a CPU ISA mode change to occur (for example, in the case of the ARM architecture, a BLX instruction) then the CPU state will be updated accordingly. Since the decoder service is a detached component, and may be called by a thread other than the main execution loop, it cannot (and should not) access the CPU state, and therefore must be instructed by the calling routine which ISA mode to use. Additionally, since a JIT compiler thread does not operate in sync with the execution thread, it also cannot access the CPU state and must call the decoder service with the ISA mode information supplied in the metadata of the basic-block it is currently compiling. A basic-block can only contain instructions of one ISA mode. This metadata is populated by the profiling element of the interpreter (see Figure 2). In order to remain retargetable (and therefore target hardware agnostic), the ISA mode is a first-class citizen in the DBT framework (see Figure 3), and is not tied to a specific architecture’s method of handling multiple ISAs. For example, the ARM architecture tracks the current ISA mode by means of the T bit in the CPSR register.

### B. Hotswapping Software Instruction Decoders

The instruction decoder is implemented as a separate component, or service, within the DBT and as such is called by any routine that requires an instruction to be decoded. Such routines would be the interpreter, when a decoder cache miss occurs, and a JIT compilation thread, when an instruction is being translated. Upon such a request being made, the decoder must be provided with the PC from which to fetch the instruction, and the ISA that the instruction should be decoded with. Given this information, as part of a decoding request, the decoder service can then make a correctly sized fetch from the guest systems memory, and select the correct decoder tree with which to perform the decode of the instruction.

The interpreter will perform the decode request using the current machine state, available as part of the execution engine, and a JIT compilation thread will perform the decode request using the snapshot of the machine state provided as part of the compilation work unit (see Figure 4).
C. High-Level Retargetability

We use a variant of the ARCHC [21] architecture description language (ADL) for the specification of the target architecture, i.e. architecturally visible registers, instruction formats and behaviours. A simplified example of our ARM V5T model is shown in Listing 1. Please note the declaration of the two supported ISAs in lines 18–19, where the system is made aware of the presence of the two target ISAs and the ARM ISA is set as a default. Within the constructor in lines 25–26 we include the detailed specifications for both supported ISA.

After the top-level model (describing register banks, registers, flags and other architectural components) has been defined, details of both supported ISAs need to be specified. Simplified examples of the ARM and THUMB ISA models are shown in Listings 2 and 3 in Figure 5. For each ISA we need to provide its name (line 4) and fetch size (line 5) (of which instruction words are multiples of). This is followed by a specification of instruction formats present in the ARM and THUMB ISAs (lines 7–11) before each instruction is assigned exactly one of the previously defined instruction formats (lines 13–17). The main sections of the instruction definitions (starting in lines 21 and 20, respectively) describe the instruction patterns for decoder generation (lines 24 and 23), their assembly patterns for disassembly (lines 25 and 24) and names of functions that implement the actual instruction semantics, also called behaviours (lines 27 and 25).

In an offline stage, we generate a target-specific processor module (see Figure 3) from this processor and ISA description. In particular, the individual decoder trees (see Figure 4) for both the ARM and THUMB ISAs are generated from an ARCHC-like specification using an approach based on [22], [23]. Note that we use ARCHC as a description language only, and do not use or implement any of the existing ARCHC tools. The benefit of choosing to use ARCHC as the description language is that it is well-known in the architecture design field, and descriptions exist for a variety of real architectures.

Listing 1: Top-level ARCHC-like specification of ARMV5T model

```
AC_ARCH(armv5t) {
  // General Purpose Registers
  ac_regbank<uint32> Rb:16;
  // General Flags
  ac_reg<uint8> C, V, Z, N;
  // Machine word size
  ac_wordsize 32;
  // Two ISAs: ARM and Thumb.
  // Make ARM the default ISA
  ac_isa_mode arm default;
  ac_isa_mode thumb;
  // Constructor
  ARCH_CTOR(armv5t) {
    // Include instruction specifications
    ac_isa("arm5t_isa.ac");
    ac_isa("armv5_thumb_isa.ac");
    // Set endiness
    set_endian("little");
  }
}
```

Listing 2: Simplified ARCHC-like specification for ARM ISA

```
AC_ISA(armv5) {
  // ARM ISA - 32bit
  ac_mode arm;
  ac_fetchsize 32;
  // Declare Instruction Formats
  ac_format Type_DPI1 =
      "%cond:4 %op:3 %func1!:4 %rs:4 %rd:4
       %shift_amt:5 %shift_type:2 %subop1!:1 %rm:4";
  // Declare Instructions
  ac_instr<Type_DPI1> adcl, ...
  ISA_CTOR(armv5) {
    // arm1 instruction
    adcl.set_decoder(op=0x00, subop1=0x00, func1=0x00);
    adcl.set_asm("adcl %reg, %reg", rd, rs);
    adcl.set_decoder(op=0x00, subop1=0x00, func1=0x00);
    adcl.set_asm("adcl %reg, %reg", rd, rs);
    adcl.set_behaviour(arm1);
  }
}
```

Listing 3: Simplified ARCHC-like specification for THUMB ISA

Fig. 5: Overview of ARCHC-like specifications for both the ARM and THUMB ISAs.

Furthermore, the instruction behaviours we define are purely semantic and are not tied to the execution pipeline.

Unlike QEMU, where instruction behaviours are expressed using sequences of calls to its low-level tiny code generator (TCG), we use high-level C code to directly express these behaviours. The advantage is in the reduced effort for retargeting to another target ISA, which in our system essentially involves copying pseudo-code instruction specifications from the processor manual into a slightly more formal C representation. Examples of both ARCHC-like and TCG semantic actions for the same ARM V5 adcl instruction are shown in Figure 6. While the ARCHC-like specification is high-level and has been directly derived from the processor manual its QEMU counterpart is low-level, complex and prone to errors.

Our generator system parses the instruction behaviours, generates an SSA form for optimisation and then generates a function that when invoked will emit L1VM bytecode for the given decoded instruction. This technique ensures only bytecode that is required for the instruction is generated, eliminating any
the profiling information (which includes a control-flow graph) encountered and after a certain configurable threshold is met, it builds profiling information about the basic blocks it has encountered as it is encountered. As the interpreter executes, instruction are used directly by the interpreter to execute each instruction as it is encountered. As the high-level implementation of the instructions are written in a strict subset of C, the behaviours for each instruction as it is encountered. As the interpreter executes, it builds profiling information about the basic blocks it has encountered and after a certain configurable threshold is met, the profiling information (which includes a control-flow graph) can be used to improve the code size of the generated code.

As the high-level implementation of the instructions are written in a strict subset of C, the behaviours for each instruction are used directly by the interpreter to execute each instruction as it is encountered. As the interpreter executes, it builds profiling information about the basic blocks it has encountered and after a certain configurable threshold is met, the profiling information (which includes a control-flow graph) can be used to improve the code size of the generated code.

unnecessary runtime decoding checks (such as flag setting). The generated processor module is dynamically loaded by our DBT system on startup and contains both a threaded interpreter and an LLVM based JIT compiler. At runtime the JIT compiler performs translation of regions of target instructions [2] to native code of the host machine using the offline generated generator functions, which employ additional dynamic optimisations such as partial evaluation [24] to improve both quality and code size of the generated code.

As the high-level implementation of the instructions are written in a strict subset of C, the behaviours for each instruction are used directly by the interpreter to execute each instruction as it is encountered. As the interpreter executes, it builds profiling information about the basic blocks it has encountered and after a certain configurable threshold is met, the profiling information (which includes a control-flow graph) can be used to improve the code size of the generated code.

unnecessary runtime decoding checks (such as flag setting). The generated processor module is dynamically loaded by our DBT system on startup and contains both a threaded interpreter and an LLVM based JIT compiler. At runtime the JIT compiler performs translation of regions of target instructions [2] to native code of the host machine using the offline generated generator functions, which employ additional dynamic optimisations such as partial evaluation [24] to improve both quality and code size of the generated code.

As the high-level implementation of the instructions are written in a strict subset of C, the behaviours for each instruction are used directly by the interpreter to execute each instruction as it is encountered. As the interpreter executes, it builds profiling information about the basic blocks it has encountered and after a certain configurable threshold is met, the profiling information (which includes a control-flow graph) can be used to improve the code size of the generated code.

unnecessary runtime decoding checks (such as flag setting). The generated processor module is dynamically loaded by our DBT system on startup and contains both a threaded interpreter and an LLVM based JIT compiler. At runtime the JIT compiler performs translation of regions of target instructions [2] to native code of the host machine using the offline generated generator functions, which employ additional dynamic optimisations such as partial evaluation [24] to improve both quality and code size of the generated code.

As the high-level implementation of the instructions are written in a strict subset of C, the behaviours for each instruction are used directly by the interpreter to execute each instruction as it is encountered. As the interpreter executes, it builds profiling information about the basic blocks it has encountered and after a certain configurable threshold is met, the profiling information (which includes a control-flow graph) can be used to improve the code size of the generated code.

unnecessary runtime decoding checks (such as flag setting). The generated processor module is dynamically loaded by our DBT system on startup and contains both a threaded interpreter and an LLVM based JIT compiler. At runtime the JIT compiler performs translation of regions of target instructions [2] to native code of the host machine using the offline generated generator functions, which employ additional dynamic optimisations such as partial evaluation [24] to improve both quality and code size of the generated code.

As the high-level implementation of the instructions are written in a strict subset of C, the behaviours for each instruction are used directly by the interpreter to execute each instruction as it is encountered. As the interpreter executes, it builds profiling information about the basic blocks it has encountered and after a certain configurable threshold is met, the profiling information (which includes a control-flow graph) can be used to improve the code size of the generated code.

unnecessary runtime decoding checks (such as flag setting). The generated processor module is dynamically loaded by our DBT system on startup and contains both a threaded interpreter and an LLVM based JIT compiler. At runtime the JIT compiler performs translation of regions of target instructions [2] to native code of the host machine using the offline generated generator functions, which employ additional dynamic optimisations such as partial evaluation [24] to improve both quality and code size of the generated code.

As the high-level implementation of the instructions are written in a strict subset of C, the behaviours for each instruction are used directly by the interpreter to execute each instruction as it is encountered. As the interpreter executes, it builds profiling information about the basic blocks it has encountered and after a certain configurable threshold is met, the profiling information (which includes a control-flow graph) can be used to improve the code size of the generated code.
We use MiPS (Millions of Instructions per-second) as a metric to measure the execution rate of both our D and Q implementations. This overhead can be attributed to the extra instructions required in T implementations. This overhead leads to a longer running time. But, the throughput of the single-I implementation is greater than that of the dual-IA implementation. Whilst the actual running time is longer for dual-I implementations, the instruction execution rate is that of the dual-I implementation. It has been shown (e.g. in [25]) that the instruction count in Table III, show that the amount of overhead is exactly the same between D and T implementations, but this figure varies significantly in QEMU. We employ a region-based compilation strategy.

The optimisation strategies employed in our D system lead to a lot of overhead. Local branches (i.e. branches within a region) are only optimised using LVM switching, and are handled by the D system through use of redundant load and store instruction.

### B. Key Results

#### 1. Data processing instructions can only operate on the first eight registers (0 to 7). Data must be explicitly moved from the high registers to the low registers. Instructions are conditionally compiled (except for the conditional branch instruction). Local branches (i.e. branches within a region) are optimised using LVM switching, and are handled by the D system through use of redundant load and store instructions. Where conditional code must be made to avoid conflicts, the appropriate predicate is inserted to exclude them from execution.

#### 2. No FLTM/SA/EMU instructions are predicted (except for the conditional branch instruction). Instructions are conditionally compiled (except for the conditional branch instruction). Local branches (i.e. branches within a region) are optimised using LVM switching, and are handled by the D system through use of redundant load and store instructions. Where conditional code must be made to avoid conflicts, the appropriate predicate is inserted to exclude them from execution.

#### C. Dynamic ISA Switching

Our results on dynamic ISA switching are summarised in Table III. For each benchmark, we list the total number of instructions (both have approximately 99%) but in nearly every case the relative execution rate of a dual-SA execution in comparison to single-SA execution is 192% faster for dual-SA.

### Table III. Summary of dynamic instruction and ISA switching counts for ARM/Thumb Single Core 2006 integer benchmarks.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Single IA</th>
<th>Dual IA</th>
<th>Single SA</th>
<th>Dual SA</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>284175873</td>
<td>40293793</td>
<td>436245530</td>
<td>481408877</td>
</tr>
<tr>
<td>gcc</td>
<td>254579753</td>
<td>382630838</td>
<td>448133895</td>
<td>507929278</td>
</tr>
<tr>
<td>gcc</td>
<td>303649472</td>
<td>213268081</td>
<td>282322663</td>
<td>309264530</td>
</tr>
<tr>
<td>gcc</td>
<td>339475820</td>
<td>1222946935</td>
<td>327246351</td>
<td>348395879</td>
</tr>
<tr>
<td>gcc</td>
<td>548796187</td>
<td>6308606485</td>
<td>485710122</td>
<td>548796187</td>
</tr>
<tr>
<td>gcc</td>
<td>270053790</td>
<td>2318531999</td>
<td>246868470</td>
<td>274587205</td>
</tr>
<tr>
<td>gcc</td>
<td>274587205</td>
<td>1092619849</td>
<td>2636610068</td>
<td>309264530</td>
</tr>
<tr>
<td>gcc</td>
<td>207064575</td>
<td>1312286829</td>
<td>309264530</td>
<td>309264530</td>
</tr>
<tr>
<td>gcc</td>
<td>287896187</td>
<td>485710122</td>
<td>548796187</td>
<td>548796187</td>
</tr>
<tr>
<td>gcc</td>
<td>131228683</td>
<td>448133895</td>
<td>548796187</td>
<td>548796187</td>
</tr>
</tbody>
</table>

### D. Comparison to State-of-the-Art

Fig. 7: Relative execution rate of dual-IA/Thumb execution in comparison to single-IA/Thumb execution.
enabling control-flow within a region to be subject to a series of loop optimisations. Our ability to hide compilation latency by means of offloading JIT compilation to multiple threads also provides a performance gain, as we are continuously executing target instructions, in contrast to QEMU which stalls as it discovers and compiles new code. The high-level code used to describe instruction implementations enables easy debugging, testing and verification, and we have internal tools that can automatically generate and run tests against reference hardware. In contrast, QEMU has a single large file that contains the decoder and the code generator, with limited documentation and no explanation of how instructions are decoded – or how their execution is modelled. Using our system, once the high-level code has been written, any improvements in the underlying framework (or even the processor module generator, see figure 3) are immediately available to all architecture descriptions, and if errors are detected in the decoder or instruction behaviours, it only requires correcting once in high-level code to fix in both the JIT and interpretive component.

E. Comparison to Native Execution

Figure 9 shows the absolute performance in target MIPS of our DBT compared with execution on a native ARM platform (QUALCOMM DRAGONBoard featuring four Snapdragon 800 cores). On average, we are 31% slower than native execution for dual-ISA implementation, but there are some cases where our simulation is actually faster than the native execution on a 1.7GHz out-of-order ARM core. For example, 429.mcf is 3.1x faster in our DBT, compared to executing natively. This may be attributed to 429.mcf warming up quite quickly in our JIT, and spending the remaining time executing host-optimised native code. Conversely, 403.gcc is 2.2x slower than native in our DBT, which may be attributed to 403.gcc’s inherently phased behaviour, and therefore invoking multiple JIT compilation sessions throughout the lifetime of the benchmark.

V. RELATED WORK

DAISY [3] is an early software dynamic translator, which uses POWERPC as the input instruction set and a proprietary VLIW architecture as the target instruction set. It does not provide for dual-mode ISA support. SHADE [4] and EMBRA [5] are DBT systems targeting the SPARC v8/v9 and MIPS I ISAS, but neither system provides support for a dual-mode ISA. STRATA [6], [7] is a retargetable software dynamic translation infrastructure designed to support experimentation with novel applications of DBT. STRATA has been used for a variety of applications including system call monitoring, profiling, and code compression. The STRATA-ARM port [8] has introduced a number of ARM-specific optimisations, for example, involving reads of and writes to the exposed PC. STRATA-ARM targets the ARM v5T ISA, but provides no support for THUMB instructions. The popular SIMPLESCALAR simulator [9] has been ported to support the ARM V4 ISA, but this port is lacking support for THUMB. The SIMIT-ARM simulator can asynchronously perform dynamic binary translation (using GCC, as opposed to an in-built translator), and accomplish this by dispatching work to other processor cores, or across the network using sockets [1]. It does not, however, support the THUMB instruction set – nor does it intend to in the near future. XTREM [10] and XEMU [11] are a power and performance simulators for the INTEL XSCLAE core. Whilst this core implements the ARM v5TE ISA, THUMB instructions are neither supported by XTREM or XEMU. FACSIM [12] is an instruction set simulator targeting the ARM9E-S family of cores, which implement the ARM v5TE architecture. FACSIM employs DBT technology for instruction-accurate simulation and interpretive simulation in its cycle-accurate mode. Unfortunately, it does not support THUMB instructions in either mode. SYNTSIM [13] is a portable functional simulator generated from a high-level architectural description. It supports the ALPHA ISA,
but provides no support for mixed-mode instruction sets. SIMICS/ARM [14] has a fairly complete implementation of the core ARM V5 instruction set. The THUMB and enhanced DSP extensions are not implemented, though. ARMSS [15] is an interpretive simulator of the ARM920T architecture, which uses instruction caching but provides no THUMB support. Similarly, the ARM port of the popular PIN tool does not support THUMB extensions [26]. As outlined above, none of the ARM DBTs mentioned support the THUMB instruction set, and others do not offer any form of multiple-ISA support specific to their target platform. This could indicate that the problem of supporting multiple instruction sets may have been deemed too complex to be worth implementing, or not yet even considered. QEMU [16] is a well-known retargetable emulator that supports ARM V5T platforms, including THUMB instructions. QEMU translates ARM/THUMB instructions to native x86 code using its tiny code generator (TCG). QEMU is interpreter-less, i.e., all executed code is translated. In particular, this means that TCG is not decoupled from the execution loop, but execution stops whilst code is JIT-compiled and only resumes afterwards. This design decision avoids the challenges outlined in this paper, but it places the JIT compiler on the critical path for code execution and misses the opportunity to offload the JIT compiler to another core of the host machine [27], [2], [28]. Another mixed-ISA simulator is presented in [29], however, this is based entirely on interpretive execution with instruction caching and about two orders of magnitude slower than either QEMU or our DBT system. ARM provides the ARMULATOR [30] and FAST MODELS [31] ISS. ARMULATOR is an interpretive ISS and has been replaced by JIT compilation-based FAST MODELS, which supports THUMB and operates at speeds comparable to QEMU-ARM, but no internal details are available due to its proprietary nature. LISA is a hardware description language aimed at describing “programmable architectures, their peripherals and interfaces”. The project also produces a series of tools that accept a LISA definition and produce a toolchain consisting of compilers, assemblers, linkers and an instruction set simulator. The simulator produced is termed a JIT-CCS (just-in-time cache compiled simulator) [32] and is a synchronous JIT-only simulator, which compiles and executes on an instruction-by-instruction basis, caching the results of the compilation for fast re-use. However, each instruction encountered is not in fact compiled as such, but rather linked to existing pre-compiled instruction behaviours as they are encountered. These links are placed in a cache, indexed by instruction address and are tagged with the instruction data. This arrangement supports self-modifying code and arbitrary ISA mode switches, as when a cache lookup occurs, the tag is checked to determine if the cached instruction is for the correct mode, and that it is equal to the one that is about to be executed. In contrast to our asynchronous approach, the simulator knows which ISA mode the emulated processor is currently in at instruction execution time and if a cache miss occurs, it can use the appropriate instruction decoder at that point to select the pre-compiled instruction implementation. As our decode and compilation phase is decoupled from the execution engine, we cannot use this method to select which decoder to use. The main drawback to this approach is that it is not strictly JIT-compilation, but rather JIT-selection of instruction implementations, and hence no kind of run-time optimisation is performed, especially since the simulation engine executes an instruction at a time. This is in contrast to our approach, which compiles an entire region of discovered guest instructions at a time, and executes within the compiled region of code. Furthermore, the instructions are only linked to behaviours, and so specialisation of the behaviours depending on static instruction fields cannot occur, resulting in greater overhead when executing an instruction. Our partial evaluation approach to instruction compilation removes this source of overhead entirely. A commercialisation of the LISA tools is available from Synopsys as their Processor Designer offering, but limited information about the implementation of
the simulators produced is available for this proprietary tool, other than an indication that it employs the same strategy as described above.

VI. SUMMARY AND CONCLUSIONS
Asynchronous mixed-mode DBT systems provide an effective means to increase JIT throughput and, at the same time, hide compilation latency, enabling the use of potentially slower, yet highly optimising code generators. In this paper we have developed a novel methodology for integrating dual-ISA support to a retargetable, asynchronous DBT system: No prior asynchronous DBT system is known to provide any support for mixed-mode ISAs. We introduce ISA mode tracking and hot-swapping of software instruction decoders as key enablers to efficient ARM/THUMB emulation. We have evaluated our approach against the SPEC CPU2006 integer benchmark suite and demonstrate that our approach to dual-ISA support does not increase any overhead. For an ARM VST model generated from a high-level description our retargetable DBT system operates at 780 MIPS on average. This is equivalent to about 192% of the performance of state-of-the-art QEMU-ARM, which has seen years of manual tuning to achieve its performance and is one of the very few DBT systems that provides both ARM and THUMB support.

REFERENCES