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

Citation for published version:
Spink, T, Wagstaff, H, Franke, B & Topham, N 2015, Efficient Dual-ISA Support in a Retargetable, Asynchronous Dynamic Binary Translator. in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on.* Institute of Electrical and Electronics Engineers (IEEE), pp. 103 - 112. https://doi.org/10.1109/SAMOS.2015.7363665

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

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 operations, 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

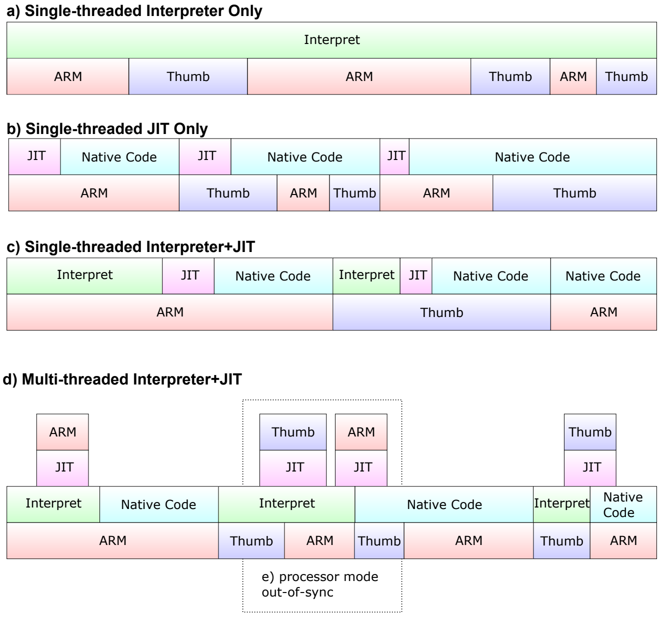

Fig. 1: Translation/Execution Models for DBT Systems.

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 archieves 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-intruction basis. Strictly, this is not DBT as no translation takes place. It is straight-forward to keep track of the current ISA as mode changing 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:

```
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:

```
mov r6, r0
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 "stripped" 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.

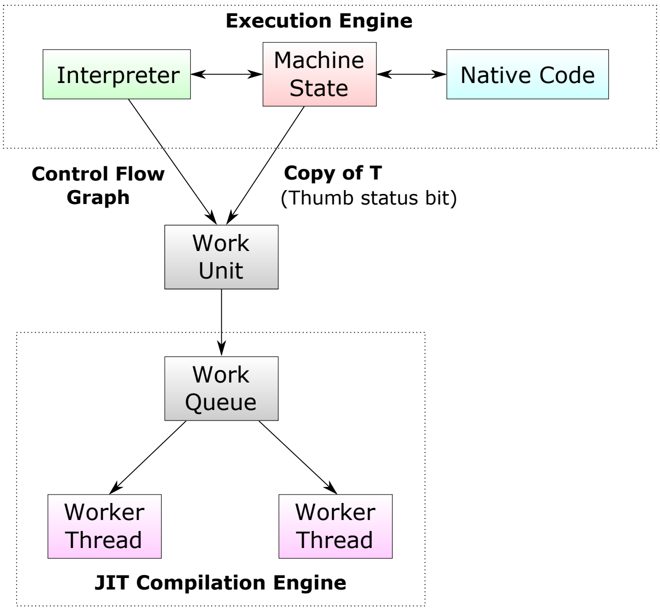

Fig. 2: The execution engine and compilation engine are distinct components and therefore cannot access the machine state. To decode correctly, a snapshot is taken of the machine state, and stored with a compilation work unit.

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

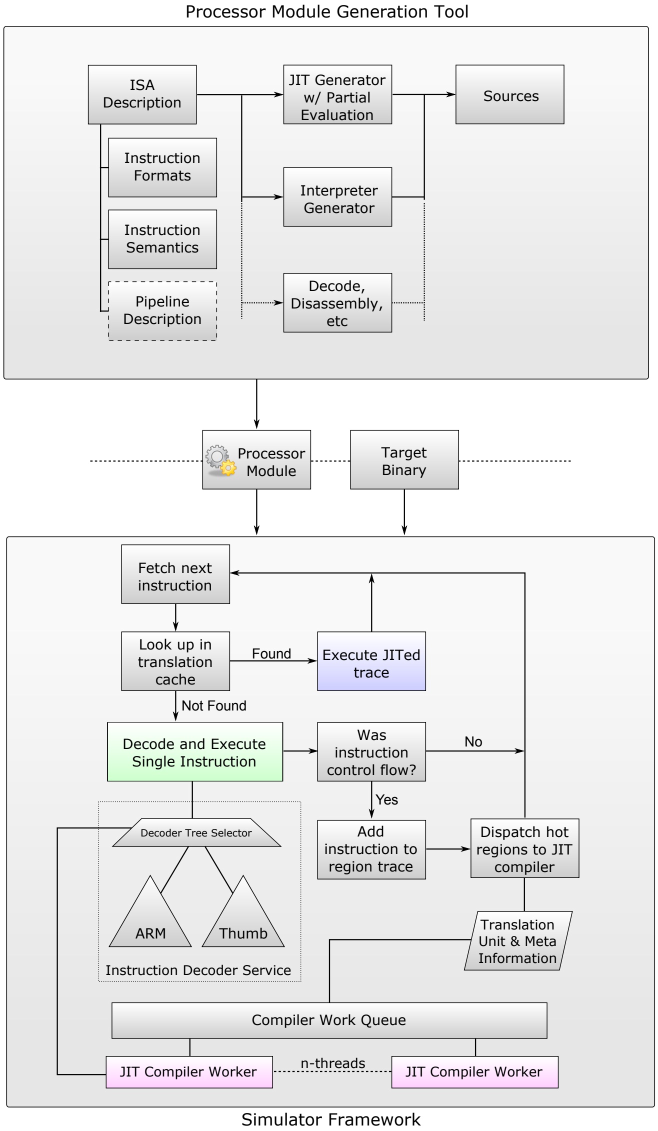

Fig. 3: Generation of a processor module from a high-level architecture description and its use within the main execution loop of our retargetable DBT system.

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.

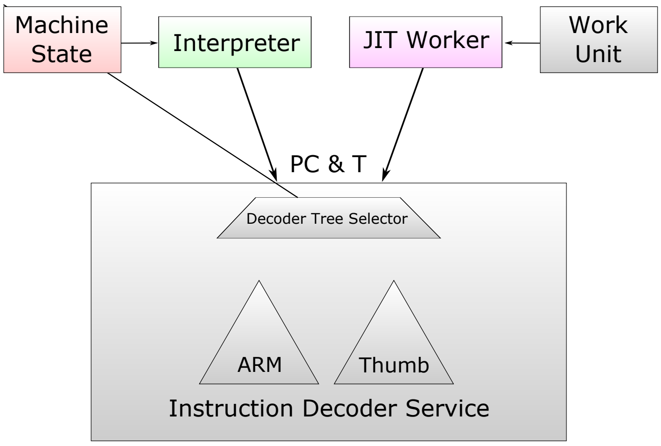

Fig. 4: Illustration of interaction between interpreter, JIT compiler and instruction decoder service.

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).

```
 1    AC_ARCH(armv5t)
 2    {
 3      // General Purpose Registers
 4      ac_regbank<uint32> RB:16;
 5
 6      // General Flags
 7      ac_reg<uint8> C, V, Z, N;
 8
 9      // Machine word size
10      ac_wordsize 32;
11
12      // Two ISAs: ARM and Thumb.
13      // Make ARM the default ISA
14      ac_isa_mode arm default;
15      ac_isa_mode thumb;
16
17      // Constructor
18      ARCH_CTOR(armv5t)
19      {
20        // Include instruction specifications
21        ac_isa("armv5_isa.ac");
22        ac_isa("armv5_thumb_isa.ac");
23
24        // Set endianess
25        set_endian("little");
26      };
27    };
```

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

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.

```
 1  AC_ISA(armv5)
 2  {
 3    // ARM ISA - 32bit
 4    ac_mode arm;
 5    ac_fetchsize 32;
 6
 7    // Declare Instruction Formats
 8    ac_format Type_DPI1 =
 9    "%cond:4 %op!:3 %func1!:4 %s:1 %rn:4 %rd:4
10     %shift_amt:5 %shift_type:2 %subop1!:1 %rm:4";
11
12    // Declare Instructions
13    ac_instr<Type_DPI1> adc1, ...
14
15    ISA_CTOR(armv5)
16    {
17      // adc1 instruction
18      adc1.set_decoder(op=0x00, subop1=0x00, func1=0x05);
19      adc1.set_asm("adc%[cond]%sf %reg, %reg, %reg",
20        cond, s, rd, rn, rm, shift_amt=0, shift_type=0);
21      adc1.set_behaviour(adc1);
22    };
23  };
```

Listing 2: Simplified ARCHC-like specification for ARM ISA

```
 1  AC_ISA(thumb)
 2  {
 3    // Thumb ISA - 16bit
 4    ac_mode thumb;
 5    ac_fetchsize 16;
 6
 7    // Declare Instruction Formats
 8    ac_format ALU_OP_INS   = "0x10:6 %op:4 %rs:3 %rd:3";
 9
10    // Declare Instructions
11    ac_instr<ALU_OP_INS> adc, ...
12
13    // Details of Thumb instructions
14    ISA_CTOR(thumb)
15    {
16      // adc instruction
17      adc.set_decoder(op=0x5);
18      adc.set_asm("adc %reg, %reg", rd, rs)
19      adc.set_behaviour(thumb_alu_adc);
20    };
21  };
```

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 `adc` 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 LLVM bitcode for the given decoded instruction. This technique ensures only bitcode that is required for the instruction is generated, eliminating any

```
/* rd = rn + imm + CF. Compute C, N, V and Z flags */
execute(adc) {
  uint32 rn_val = read_register(inst.rn);
  uint32 imm = ROR(inst.imm, inst.rot);
  uint32 c_flag = read_flag(C);

  uint32 rd_val = rn_val + imm + c_flag;
  write_register(inst.rd, rd_val);

  if(inst.S) {
    write_flag(N, !!(rd_val & 0x80000000));
    write_flag(Z, rd_val == 0);
    write_flag(C, carry_from(rn_val, imm, c_flag));
    write_flag(V, overflow_from(rn_val, imm, c_flag));
  }
}
```

Listing 4: ArchC-like behaviour for an Arm v5 `adc` instruction

```
/* dest = T0 + T1 + CF. Compute C, N, V and Z flags */
static void gen_adc_CC(TCGv_i32 dest, TCGv_i32 t0,
                       TCGv_i32 t1)
{
    TCGv_i32 tmp = tcg_temp_new_i32();
    if (TCG_TARGET_HAS_add2_i32) {
        tcg_gen_movi_i32(tmp, 0);
        tcg_gen_add2_i32(cpu_NF, cpu_CF, t0, tmp,
                         cpu_CF, tmp);
        tcg_gen_add2_i32(cpu_NF, cpu_CF, cpu_NF,
                         cpu_CF, t1, tmp);
    } else {
        TCGv_i64 q0 = tcg_temp_new_i64();
        TCGv_i64 q1 = tcg_temp_new_i64();
        tcg_gen_extu_i32_i64(q0, t0);
        tcg_gen_extu_i32_i64(q1, t1);
        tcg_gen_add_i64(q0, q0, q1);
        tcg_gen_extu_i32_i64(q1, cpu_CF);
        tcg_gen_add_i64(q0, q0, q1);
        tcg_gen_extr_i64_i32(cpu_NF, cpu_CF, q0);
        tcg_temp_free_i64(q0);
        tcg_temp_free_i64(q1);
    }
    tcg_gen_mov_i32(cpu_ZF, cpu_NF);
    tcg_gen_xor_i32(cpu_VF, cpu_NF, t0);
    tcg_gen_xor_i32(tmp, t0, t1);
    tcg_gen_andc_i32(cpu_VF, cpu_VF, tmp);
    tcg_temp_free_i32(tmp);
    tcg_gen_mov_i32(dest, cpu_NF);
}
```

Listing 5: Equivalent Qemu code for an `adc` instruction

Fig. 6: Comparison: Semantic action of an Arm v5 `adc` instruction expressed as using high-level ArchC-like specification (top) and low-level Qemu implementation (bottom).

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) is sent as a compilation work unit to the work unit queue, where it is picked up by an idle compiler worker thread. The worker thread then processes the blocks within the work unit, and (utilising the generator functions) generates native code for the block (see Figure 3).

TABLE I: Dbt Host Configuration.

Vendor & Model	Dell™PowerEdge™ R610
Processor Type	2× Intel©Xeon™ X5660
Number of cores	2 × 6
Clock/Fsb Frequency	2.80/1.33 Ghz
L1-Cache	2 × 6× 32K Instruction/Data
L2-Cache	2 × 6× 256K
L3-Cache	2× 12 Mb
Memory	36 Gb across 6 channels
Operating System	Linux version 2.6.32 (x86-64)

TABLE II: Dbt System Configuration.

DBT Parameter	**Setting**
Target architecture	Arm v5T
Host architecture	x86-64
Translation/Execution Model	Asynch. Mixed-Mode
Tracing Scheme	Region-based [2]
Tracing Interval	30000 blocks
Jit compiler	Llvm 3.4
No. of Jit Compilation Threads	10
Jit Optimisation	`-O3` & Part. Eval. [24]
Dynamic Jit Threshold	Adaptive [2]
System Calls	Emulation

IV. Experimental Evaluation

A. *Experimental Setup and Methodology*

The target architecture for our Dbt system is Arm v5T. We provide full coverage of both the standard Arm and compact Thumb Isas. The host machine we have used for performance measurements is a 12-core `x86` Dell™ PowerEdge™ as described in Table I. We have configured our Dbt system according to the information provided in Table II.

We have evaluated our retargetable Dbt system using the Spec Cpu2006 integer benchmark. It is widely used and considered to be representative of a broad spectrum of application domains. We used it together with its *reference* data sets. The benchmarks have been compiled using the Gcc 4.6.0 C/C++ cross-compilers, targeting the Arm v5T architecture (without hardware floating-point support) and enabling Thumb code generation with `-O3` optimisation settings. We have measured the elapsed real time between invocation and termination of each benchmark in our Dbt system using the Unix `time` command. We used the average elapsed wall clock time across 10 runs for each benchmark and configuration in order to calculate execution rates (using Mips in terms of target instructions) and speedups. For summary figures we report harmonic means, weighted by by dynamic instruction count, to ensure the averages account for the different running times of benchmarks. For the comparison to the state-of-the-art we use the Arm port of Qemu 1.4.2 as a baseline.

TABLE III: Summary of dynamic instruction and ISA switching counts for ARM/THUMB SPEC CPU2006 integer benchmarks.

Benchmark	Single ISA: ARM	Dual ISA: ARM/THUMB				
	Total # Instr.	Total # Instr.	# ARM Instr.	# THUMB Instr.	# ISA Switches	# Instr./ISA Sw.
400.perlbench	2070645752958	2745872053111	109261984987 (3.98%)	2636610068124 (96.02%)	3092645300	887.9
401.bzip2	2609484715134	3391051042092	3957811169 (0.12%)	3387093230923 (99.88%)	41230478	82246.2
403.gcc	1344893784628	1585255060497	264507977163 (16.69%)	1320747083334 (83.31%)	5927290776	267.5
429.mcf	331216948652	371554040583	1168734854 (0.31%)	370385305729 (99.69%)	15390338	24142.0
445.gobmk	2060618929096	2700537905311	231853199930 (8.59%)	2468684705381 (91.41%)	13122868288	205.8
456.hmmer	4178532202837	5487961870059	630860648583 (11.50%)	4857101221476 (88.50%)	18791239666	292.0
458.sjeng	2750900623655	3394758209517	122294693575 (3.60%)	3272463515942 (96.40%)	3483958798	974.4
462.libquantum	3121145754851	3036494720123	213268081860 (7.02%)	2823226638263 (92.98%)	2313765054	1312.4
464.h264ref	4362455306706	4814088772603	382630838508 (7.95%)	4431457934095 (92.05%)	10733077846	448.5
471.omnetpp	1245176341871	1368136735157	688403094229 (50.32%)	679733640928 (49.68%)	37557603952	36.4
473.astar	1208355180711	1601164683296	9017475374 (0.56%)	1592147207922 (99.44%)	497065322	3221.2
483.xalancbmk	1196441939837	1367527495851	135130437925 (9.88%)	1232397057926 (90.12%)	3559980618	384.1

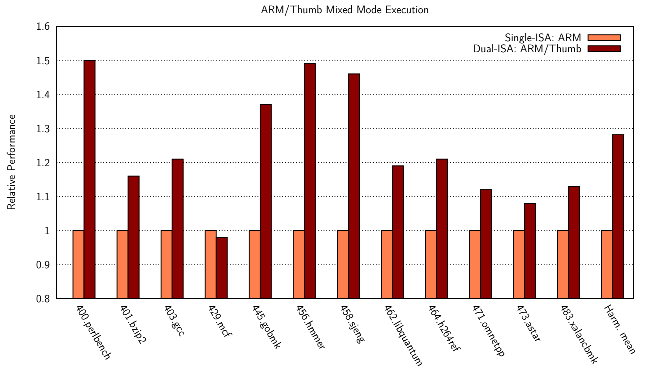

Fig. 7: Relative execution rate of dual-ARM/THUMB execution in comparison to single-ISA ARM execution.

B. *Key Results*

We use MIPS (Millions of Instructions per-second) as a metric to measure the execution rate of both our DBT and QEMU, where the instruction execution rate is that of the *target* instructions executed per-second by the DBT. Since the number of target instructions does not change between the DBT systems (as we use exactly the same binary with exactly the same input for each test in both our DBT and in QEMU), this also directly correlates to *total runtime*, but we choose to present in MIPS to show the instruction throughput, in accordance with industry practice. Figure 7 shows that in nearly every case the relative execution rate of a dual-ISA implementation of the benchmark is greater than that of the single-ISA implementation. Whilst the actual running times are longer for dual-ISA binaries (due to the higher dynamic instruction count), the DBT throughput is greater and on average we achieve a 1.28*x* improvement in execution rate over single-ISA. The instruction counts in Table III, show that more instructions are executed for dual-ISA implementations, which leads to a longer running time. But, the throughput of our DBT (as measured in target MIPS) outperforms the single-ISA implementation. It has been shown (e.g. in [25]) that whilst THUMB compiled applications are typically *physically* smaller than when compiled for ARM, the amount of overhead introduced leads to a greater execution time for actual hardware implementations. This overhead can be attributed to the extra operations required in THUMB mode, to achieve the same effect in ARM mode. Specifically, there are two main sources of overhead:

1) Data processing instructions can only operate on the first eight registers (`r0` to `r7`) - data must be explicitly moved from the high registers to the low registers.
2) No THUMB instructions (except for the conditional branch instruction) are predicated, and therefore local branches around conditional code must be made, in contrast to ARM where blocks of instructions can be simply marked-up with the appropriate predicate to exclude them from execution.

The optimisation strategies employed in our DBT system remove a lot of this overhead, local branches (i.e. branches within a region) are heavily optimised using standard LLVM optimisation passes and high-register operations are negated through use of redundant-load and dead-store elimination.

C. *Dynamic* ISA *Switching*

Our results on dynamic ISA switching are summarised in Table III. For each benchmark we list the total number of ARM instructions, ARM/THUMB instructions, ISA switches and average dynamic instruction count between ISA switches. All benchmarks make use of both the ARM and THUMB ISAs. On average 8.76% of the total number of instructions are ARM, the rest THUMB instructions, but this figure varies significantly between benchmarks. 401.bzip2 and 429.mcf have similar ratios of THUMB instructions (both have approximately 99%) but quite different relative performance characteristics. 429.mcf executes 3% slower in dual-ISA mode, where 401.bzip2 executes 16% faster. This kind of variance indicates that our DBT supporting a dual-ISA does not necessarily introduce any overhead, but is simply a function of the behaviour of the binary being translated.

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

Figure 8 shows the absolute performance in *target* MIPS of our DBT compared with the state-of-the-art QEMU. The performance of our DBT system is consistently higher than that of QEMU, on average our DBT is 192% faster for dual-ISA implementations. Since the target instruction count is exactly the same between DBTs (per benchmark), this also indicates an improvement in DBT running time. We can attribute this to the ability of our JIT compiler to produce highly optimised native code, using aggressive LLVM optimisations that simply do not (and can not, given the trace-based architecture) exist in QEMU. We employ a region-based compilation strategy,

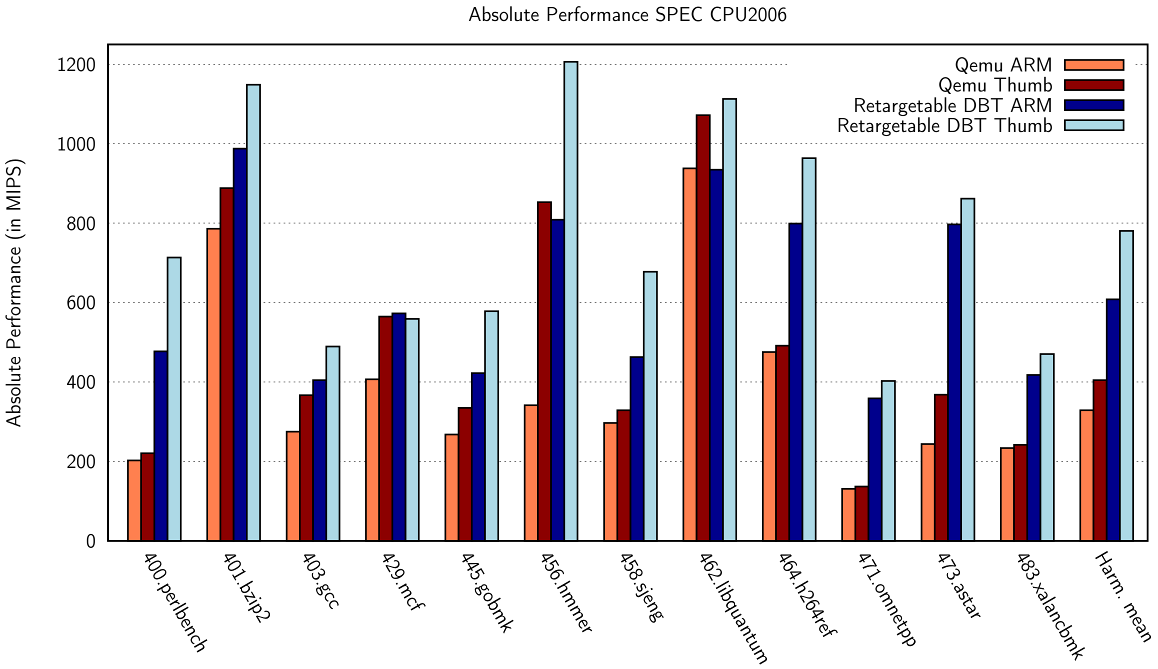

Fig. 8: Absolute performance (in target MIPS) of single- and mixed-mode execution in our retargetable DBT and QEMU.

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 3.1*x* 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 2.2*x* 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 1 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 XEEMU [11] are a power and performance simulators for the Intel XScale core. Whilst this core implements the ARM v5TE ISA, Thumb instructions are neither supported by XTrem or XEEMU. 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,

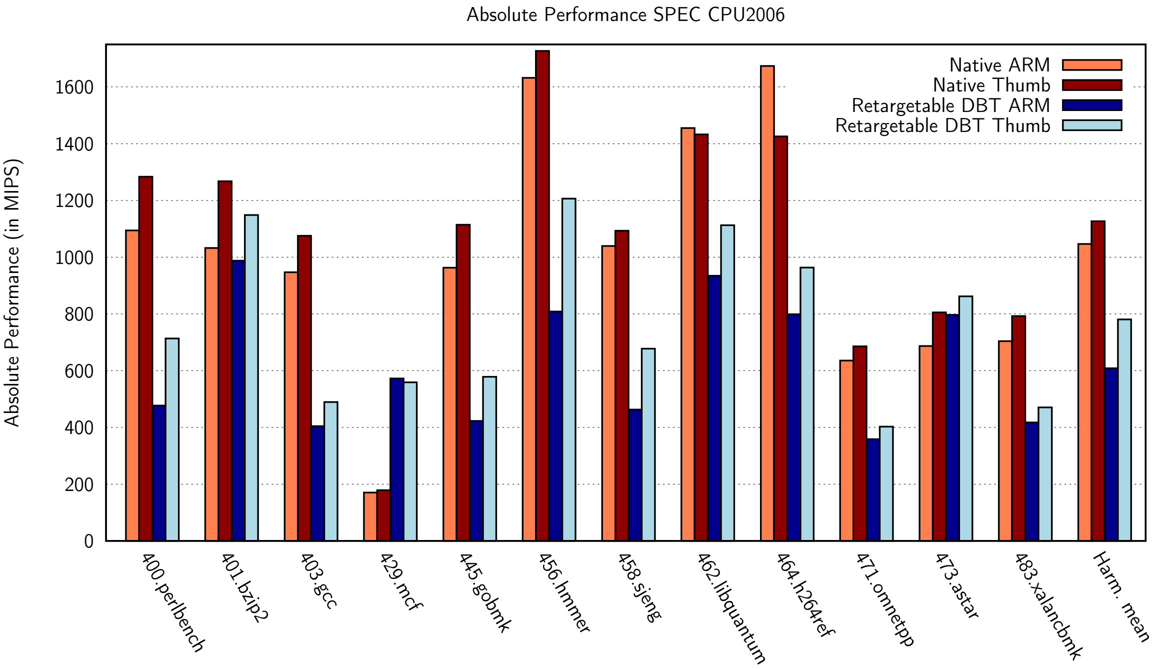

Fig. 9: Absolute performance (in target MIPS) of single- and mixed-mode execution in native execution and our retargetable DBT.

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. ARMISS [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 introduce any overhead. For an ARM V5T 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

[1] W. Qin, "SimIt-ARM 3.0," 2007.

[2] I. Böhm, T. J. Edler von Koch, S. C. Kyle, B. Franke, and N. Topham, "Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 74–85.

[3] K. Ebcioğlu and E. R. Altman, "DAISY: dynamic compilation for 100% architectural compatibility," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 26–37.

[4] B. Cmelik and D. Keppel, "Shade: A fast instruction-set simulator for execution profiling," in *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 128–137.

[5] E. Witchel and M. Rosenblum, "Embra: fast and flexible machine simulation," in *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp. 68–79.

[6] K. Scott and J. Davidson, "Strata: A software dynamic translation infrastructure," in *In IEEE Workshop on Binary Translation*, 2001.

[7] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa, "Retargetable and reconfigurable software dynamic translation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, pp. 36–47.

[8] R. W. Moore, J. A. Baiocchi, B. R. Childers, J. W. Davidson, and J. D. Hiser, "Addressing the challenges of DBT for the ARM architecture," in *Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pp. 147–156.

[9] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, Feb. 2002.

[10] G. Contreras, M. Martonosi, J. Peng, G.-Y. Lueh, and R. Ju, "The XTREM power and performance simulator for the Intel XScale core: Design and experiences," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 1, Feb. 2007.

[11] Z. Herczeg, A. Kiss, D. Schmidt, N. Wehn, and T. Gyimóthy, "XEEMU: An Improved XScale Power Simulator," in *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, ser. Lecture Notes in Computer Science, N. AzÃl'mard and L. Svensson, Eds. Springer Berlin Heidelberg, 2007, vol. 4644, pp. 300–309.

[12] 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 *Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pp. 89–100.

[13] M. Burtscher and I. Ganusov, "Automatic synthesis of high-speed processor simulators," in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 55–66.

[14] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

[15] M. Lv, Q. Deng, N. Guan, Y. Xie, and G. Yu, "ARMISS: An instruction set simulator for the ARM architecture," in *International Conference on Embedded Software and Systems, 2008. ICESS '08.*, 2008, pp. 548–555.

[16] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, pp. 41–41.

[17] N. Jia, C. Yang, J. Wang, D. Tong, and K. Wang, "SPIRE: improving dynamic binary translation through SPC-indexed indirect branch redirecting," in *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 1–12.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 190–200.

[19] F. Brandner, A. Fellnhofer, A. Krall, and D. Riegler, "Fast and accurate simulation using the LLVM compiler framework," in *Proceedings of the 1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, January 2009.

[20] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: a transparent dynamic optimization system," in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pp. 1–12.

[21] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros, "The ArchC architecture description language and tools," *International Journal of Parallel Programming*, vol. 33, no. 5, pp. 453–484, 2005.

[22] R. Krishna and T. Austin, "Efficient software decoder design," in *Proceedings of the 2001 Workshop on Binary Translation*, September 2001.

[23] W. Qin and S. Malik, "Automated synthesis of efficient binary decoders for retargetable software toolkits," in *Proceedings of the 2003 Design Automation Conference*, 2003, pp. 764–769.

[24] H. Wagstaff, M. Gould, B. Franke, and N. Topham, "Early partial evaluation in a JIT-compiled, retargetable instruction set simulator generated from a high-level architecture description," in *Proceedings of the Annual Design Automation Conference*, pp. 21:1–21:6.

[25] A. Krishnaswamy and R. Gupta, "Profile guided selection of ARM and Thumb instructions," in *Proceedings of the joint conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, pp. 56–64.

[26] K. Hazelwood and A. Klauser, "A dynamic binary instrumentation engine for the ARM architecture," in *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pp. 261–270.

[27] M. P. Plezbert and R. K. Cytron, "Does "just in time" = "better late than never"?" in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 120–131.

[28] P. A. Kulkarni, "JIT compilation policy for modern machines," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pp. 773–788.

[29] T. Stripf, R. Koenig, and J. Becker, "A cycle-approximate, mixed-ISA simulator for the KAHRISMA architecture," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012, pp. 21–26.

[30] ARM Ltd., "The ARMulator," 2003.

[31] ——, "Fast Models," 2013.

[32] A. Nohl, G. Braun, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr, "A universal technique for fast and flexible instruction-set architecture simulation," in *Proceedings of the Design Automation Conference (DAC)*, June 2002.