Cycle-Accurate Performance Modelling in an Ultra-Fast Just-In-Time Dynamic Binary Translation Instruction Set Simulator

Igor Böhm, Björn Franke, and Nigel Topham

Institute for Computing Systems Architecture,
School of Informatics, University of Edinburgh
Informatics Forum, 10 Crichton Street, Edinburgh, EH8 9AB, United Kingdom
I.Bohm@sms.ed.ac.uk, {bfranke, npt}@inf.ed.ac.uk
http://groups.inf.ed.ac.uk/pasta/

Abstract. Instruction set simulators (ISS) are vital tools for compiler and processor architecture design space exploration and verification. State-of-the-art simulators using just-in-time (JIT) dynamic binary translation (DBT) techniques are able to simulate complex embedded processors at speeds above 500 MIPS. However, these *functional* ISS do not provide microarchitectural observability. In contrast, low-level *cycle-accurate* ISS are too slow to simulate full-scale applications, forcing developers to revert to FPGA-based simulations. In this paper we demonstrate that it is possible to run ultra-high speed *cycle-accurate* instruction set simulations surpassing FPGA-based simulation speeds. We extend the JIT DBT engine of our ISS and augment JIT generated code with a verified cycle-accurate processor model. Our approach can model any microarchitectural configuration, does not rely on prior profiling, instrumentation, or compilation, and works for all binaries targeting a state-of-the-art embedded processor implementing the ARCompact™ instruction set architecture (ISA). We achieve simulation speeds up to 88 MIPS on a standard x86 desktop computer for the industry standard EEMBC, COREMARK and BIOPERF benchmark suites.

1 Introduction

Simulators play an important role in the design of today's high performance microprocessors. They support design-space exploration, where processor characteristics such as speed and power consumption are accurately predicted for different architectural models. The information gathered enables designers to select the most efficient processor designs for fabrication. On a slightly higher level instruction set simulators provide a platform on which experimental instruction set architectures can be tested, and new compilers and applications may be developed and verified. They help to reduce the overall development time for new microprocessors by allowing concurrent engineering during the design phase. This is especially important for embedded system-on-chip (SOC) designs, where processors may be extended to support specific applications. However, increasing size and complexity of embedded applications challenges current ISS technology. For example, the JPEG encode and decode EEMBC benchmarks execute between $10 * 10^9$ and $16 * 10^9$ instructions. Similarly, AAC (Advanced Audio Coding)

decoding and playback of a six minute excerpt of Mozart's *Requiem* using a sample rate of 44.1 kHz and a bit rate of 128 kbps results in $\approx 38 * 10^9$ executed instructions. These figures clearly demonstrate the need for fast ISS technology to keep up with performance demands of real-world embedded applications.

The broad introduction of multi-core systems, e.g. in the form of multi-processor systems-on-chip (MPSOC), has exacerbated the strain on simulation technology and it is widely acknowledged that improved single-core simulation performance is key to making the simulation of larger multi-core systems a viable option [1].

This paper is concerned with ultra-fast ISS using recently developed just-in-time (JIT) dynamic binary translation (DBT) techniques [27,6,15]. DBT combines interpretive and compiled simulation techniques in order to maintain high speed, observability and flexibility. However, achieving accurate state and even more so microarchitectural observability remains in tension with high speed simulation. In fact, none of the existing JIT DBT ISS [27,6,15] maintains a detailed performance model.

In this paper we present a novel methodology for *fast* and *cycle-accurate* performance modelling of the processor pipeline, instruction and data caches, and memory within a JIT DBT ISS. Our main contribution is a simple, yet powerful software pipeline model together with an instruction operand dependency and side-effect analysis JIT DBT pass that allows to retain an ultra-fast *instruction-by-instruction* execution model without compromising microarchitectural observability. The essential idea is to reconstruct the microarchitectural pipeline state *after* executing an instruction. This is less complex in terms of runtime and implementation than a *cycle-by-cycle* execution model and reduces the work for pipeline state updates by more than an order of magnitude.

In our ISS we maintain additional data structures relating to the processor pipeline and the caches and emit lightweight calls to functions updating the processor state in the JIT generated code. In order to maintain flexibility and to achieve high simulation speed our approach decouples the performance model in the ISS from the functional simulation, thereby eliminating the need for extensive rewrites of the simulation framework to accommodate microarchitectural changes. In fact, the strict separation of concerns (functional simulation *vs.* performance modelling) enables the automatic generation of a pipeline performance model from a processor specification written in an architecture description language (ADL) such as LISA [26]. This is, however, beyond the scope of this paper.

We have evaluated our performance modelling methodology against the industry standard EEMBC, COREMARK, and BIOPERF benchmark suites for our ISS of the ENCORE [33] embedded processor implementing the ARCompact™ [32] ISA. Our ISS faithfully models the 5-stage interlocked ENCORE processor pipeline (see Figure 3) with forwarding logic, its mixed-mode 16/32-bit instruction set, zero overhead loops, static and dynamic branch prediction, branch delay slots, and four-way set associative data and instruction caches. We also provide results for the 7-stage ENCORE processor pipeline variant modelled by our ISS. Across all 44 benchmarks from EEMBC, COREMARK, and BIOPERF the speed of simulation reaches up to 88 MIPS on a standard `x86` desktop computer and outperforms that of a speed-optimised FPGA implementation of the ENCORE processor.

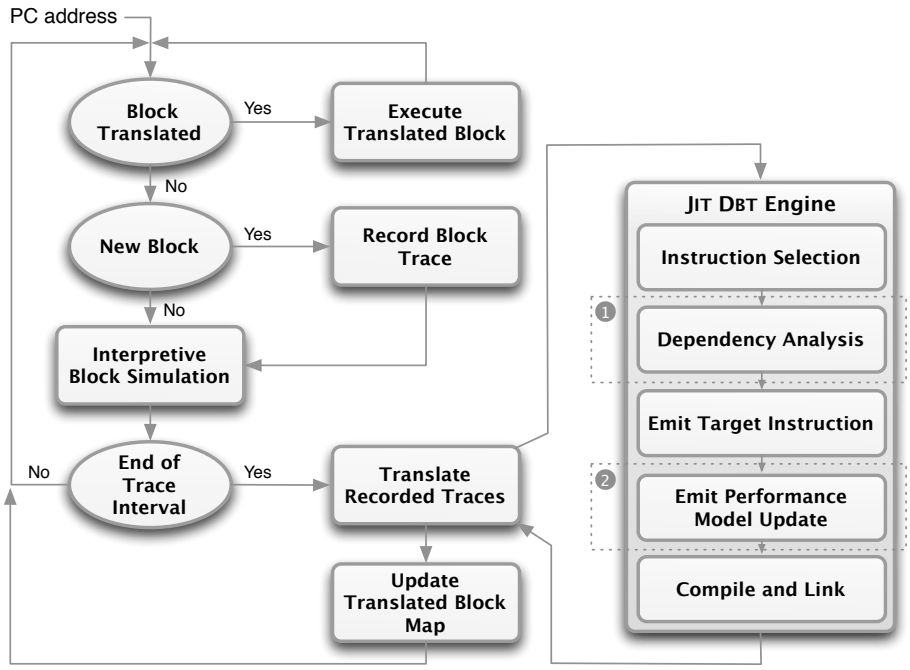

Fig. 1. Dynamic binary translation flow integrated into main simulation loop.

1.1 Motivating Example

Before we take a more detailed look at our JIT DBT engine and the proposed JIT performance model code generation approach, we provide a motivating example in order to highlight the key concepts.

Consider the block of ARCompact™ instructions in Figure 2 taken from the COREMARK benchmark. Our ISS identifies this block of code as a hotspot and compiles it to native machine code using the sequence of steps illustrated in Figure 1. Each block maps onto a function denoted by its address (see label ① in Figure 2), and each instruction is translated into semantically equivalent native code faithfully modelling the processors architectural state (see labels ②, ③, and ⑥ in Figure 2). In order to correctly track microarchitectural state, we augment each translated ARCompact™ instruction with calls to specialised functions (see labels ③ and ⑦ in Figure 2) responsible for updating the underlying microarchitectural model (see Figure 3).

Figure 3 demonstrates how the hardware pipeline microarchitecture is mapped onto a software model capturing its behaviour. To improve the performance of microarchitectural state updates we emit several versions of performance model update functions tailored to each instruction *kind* (i.e. arithmetic and logical instructions, load/store instructions, branch instructions). Section 3.1 describes the microarchitectural software model in more detail.

After code has been emitted for a batch of blocks, it is translated and linked by a JIT compiler. Finally, the translated block map is updated with addresses of each newly translated block. On subsequent encounters to a previously translated block during simulation, it will be present in the translated block map and can be executed directly.

1.2 Contributions

Among the contributions of this paper are:

1. The development of a cycle-accurate timing model for state-of-the-art embedded processors that can be adapted to different microarchitectures and is independent of the implementation of a functional ISS,
2. the integration of this cycle-accurate timing model into a JIT DBT engine of an ISS to improve the speed of cycle-accurate instruction set simulation to a level that is higher than a speed-optimised FPGA implementation of the same processor core, without compromising accuracy,
3. an extensive evaluation against industry standard COREMARK, EEMBC, and BIOPERF benchmark suites and an interpretive cycle-accurate mode of our ISS that has been verified and calibrated against an actual state-of-the-art hardware implementation of the ENCORE embedded processor implementing the full ARCompact™ ISA.

1.3 Overview

The remainder of this paper is structured as follows. In section 2 we provide a brief outline of the ENCORE embedded processor that serves as a simulation target in this paper. In addition, we outline the main features of our ARCSIM ISS and describe the basic functionality of its JIT DBT engine. This is followed by a description of our approach to decoupled, cycle-accurate performance modelling in the JIT generated code in section 3. We present the results of our extensive, empirical evaluation in section 4 before we discuss the body of related work in section 5. Finally, we summarise and conclude in section 6.

2 Background

2.1 The ENCORE Embedded Processor

In order to demonstrate the effectiveness of our approach we use a state-of-the-art processor implementing the ARCompact™ ISA, namely the ENCORE [33].

The ENCORE's microarchitecture is based on an interlocked pipeline with forwarding logic, supporting zero overhead loops (ZOL), freely intermixable 16- and 32-bit instruction encodings, static and dynamic branch prediction, branch delay slots, and predicated instructions. There exist two pipeline variants of the ENCORE processor, namely a 5-stage (see Figure 3) variant and a 7-stage variant which has an additional ALIGN stage between the FETCH and DECODE stages, and an additional REGISTER stage between the DECODE and EXECUTE stages.

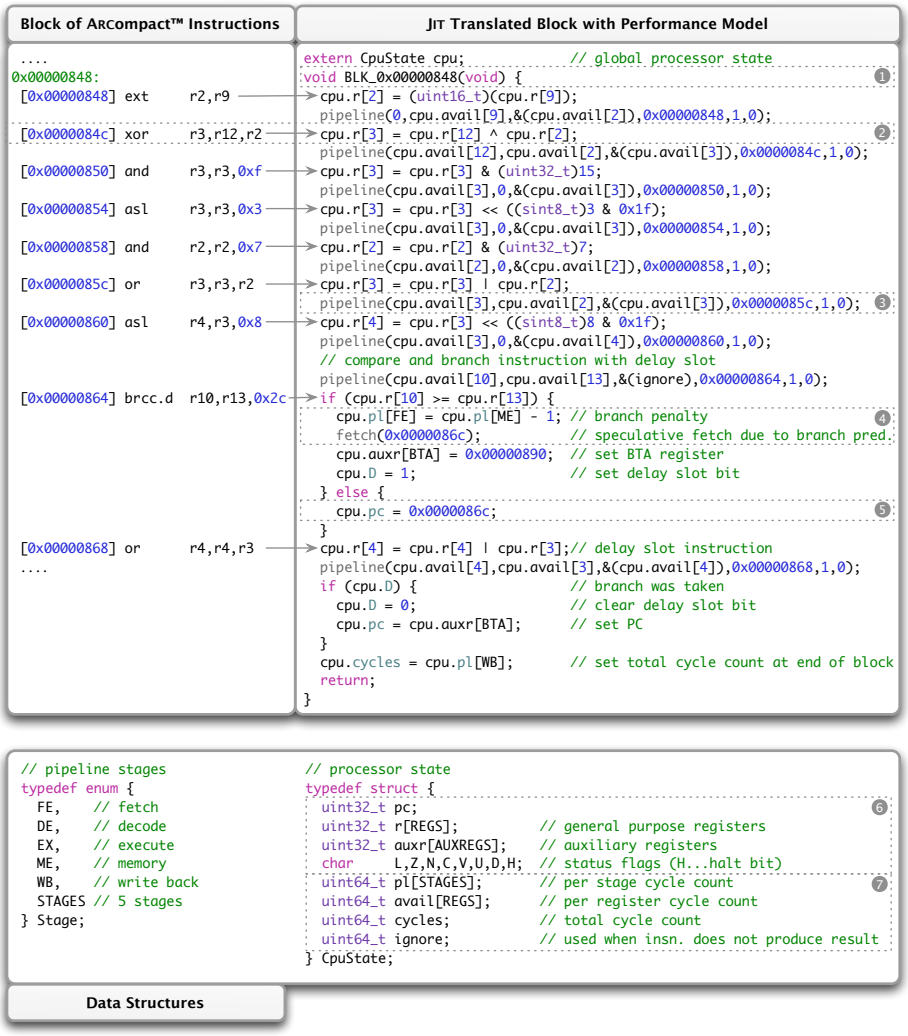

Fig. 2. JIT dynamic binary translation of ARCompact™ basic block with `CpuState` structure representing architectural ⑥ and microarchitectural state ⑦. See Figure 3 for an implementation of the microarchitectural state update function `pipeline()`.

In our configuration we use 32K 4-way set associative instruction and data caches with a pseudo-random block replacement policy. Because cache misses are expensive, a pseudo-random replacement policy requires us to *exactly* model cache behaviour to avoid large deviations in cycle count. Although the above configuration was used for this work, the processor is highly configurable. Pipeline depth, cache sizes, associativity, and block replacement policies as well as byte order (i.e. big endian, little endian), bus widths, register-file size, and instruction set specific options such as instruction set extensions (ISEs) are configurable. The processor is fully synthesisable onto an FPGA and fully working ASIP silicon implementations have been taped-out recently.

2.2 ARCSIM Instruction Set Simulator

In our work we extended ARCSIM [34], a target adaptable simulator with extensive support of the ARCompact™ ISA. It is a full-system simulator, implementing the processor, its memory sub-system (including MMU), and sufficient interrupt-driven peripherals to simulate the boot-up and interactive operation of a complete Linux-based system. The simulator provides the following simulation modes:

- *Co-simulation* mode working in lock-step with standard hardware simulation tools used for hardware and performance verification.
- Highly-optimised [27] *interpretive* simulation mode.
- Target microarchitecture adaptable *cycle-accurate* simulation mode modelling the processor pipeline, caches, and memories. This mode has been calibrated against a 5-stage and 7-stage pipeline variant of the ENCORE processor.
- *High-speed* JIT DBT functional simulation mode [27][15] capable of simulating an embedded system at speeds approaching or even exceeding that of a silicon ASIP whilst faithfully modelling the processor's architectural state.
- A *profiling* simulation mode that is orthogonal to the above modes delivering additional statistics such as dynamic instruction frequencies, detailed per register access statistics, per instruction latency distributions, detailed cache statistics, executed delay slot instructions, as well as various branch predictor statistics.

In common with the ENCORE processor, the ARCSIM simulator is highly configurable. Architectural features such as register file size, instruction set extensions, the set of branch conditions, the auxiliary register set, as well as memory mapped IO extensions can be specified via a set of well defined APIs and configuration settings. Furthermore, microarchitectural features such as pipeline depth, per instruction execution latencies, cache size and associativity, cache block replacement policies, memory subsystem layout, branch prediction strategies, as well as bus and memory access latencies are fully configurable. The microarchitectural configurations used for our experiments are listed in Table 2.

2.3 Hotspot Detection and JIT Dynamic Binary Translation

In ARCSIM simulation time is partitioned into *epochs*, where each epoch is defined as the interval between two successive JIT translations. Within an epoch frequently

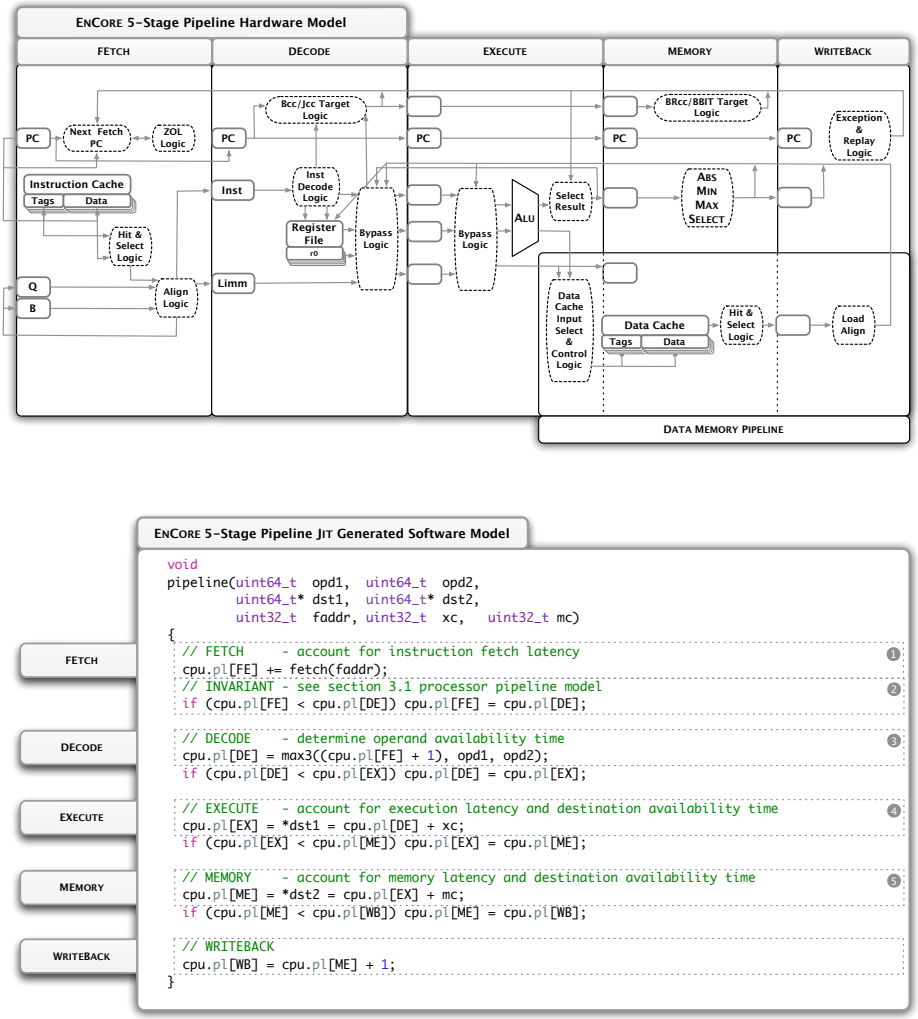

Fig. 3. EnCore 5-Stage hardware pipeline model with a sample JIT generated software microarchitectural model.

executed basic blocks (i.e. hotspots) are detected at runtime and recorded as traces (see Figure 1). After each epoch the hottest recorded traces (i.e. frequently executed traces) are passed to the JIT DBT engine for native code generation. More recently [15] we have extended hotspot detection and JIT DBT with the capability to find and translate large translation units (LTU) consisting of multiple traced control-flow-graphs. By increasing the size of translation units it is possible to achieve significant speedups in simulation performance. The simulation speedup can be attributed to improved locality, more time is spent simulating *within* a translation unit, and greater scope for optimisations for the JIT compiler as it can optimise across multiple blocks.

3 Methodology

In this paper we describe our approach to combine *cycle accurate* and *high-speed* JIT DBT simulation modes in order to provide architectural and microarchitectural observability at speeds exceeding speed-optimised FPGA implementations. We do this by extending our JIT DBT engine with a pass responsible for analysing instruction operand dependencies and side-effects, and an additional code emission pass emitting specialised code for performance model updates (see labels ① and ② in Figure 1).

In the following sections we outline our generic processor pipeline model and describe how to account for instruction operand availability and side-effect visibility timing. We also discuss our cache and memory model and show how to integrate control flow and branch prediction into our microarchitectural performance model.

3.1 Processor Pipeline Model

The granularity of execution on hardware and RTL simulation is cycle based —*cycle-by-cycle*. If the designer wants to find out how many cycles it took to execute an instruction or program, all that is necessary is to simply count the number of cycles. While this execution model works well for hardware it is too detailed and slow for ISS purposes. Therefore fast functional ISS have an *instruction-by-instruction* execution model. While this execution model yields faster simulation speeds it usually compromises microarchitectural observability and detail. Our software pipeline model together with an instruction operand dependency and side-effect analysis JIT DBT pass allows to retain an *instruction-by-instruction* execution model without compromising microarchitectural observability. The essential idea is to reconstruct the microarchitectural pipeline state *after* executing an instruction.

Thus the processor pipeline is modelled as an array with as many elements as there are pipeline stages (see definition of `pl[STAGES]` at label ⑦ in Figure 2). For each pipeline stage we add up the corresponding latencies and store the cycle-count at which the instruction is ready to *leave* the respective stage. The line with label ① in Figure 3 demonstrates this for the fetch stage `cpu.pl[FE]` by adding the amount of cycles it takes to fetch the corresponding instruction to the current cycle count at that stage. The next line in Figure 3 with the label ② is an *invariant* ensuring that an instruction cannot leave its pipeline stage *before* the instruction in the immediately following stage is ready to proceed. Figure 4 contains a detailed example of the microarchitectural performance model determining the cycle count for a sample ARCompact™ instruction.

	FE	DE	EX	ME	WB	
Instruction	or r4,r4,r3					Pipeline Model
FETCH	100	102	103	110	115	`// INITIAL STATE AT FETCH`
	101	102	103	110	115	`cpu.pl[FE] += fetch(0x00000868);`
	102	102	103	110	115	`if (cpu.pl[FE] < cpu.pl[DE])` `  cpu.pl[FE] = cpu.pl[DE];`
DECODE	102	102	103	110	115	`// INITIAL STATE AT DECODE`
	102	105	103	110	115	`cpu.pl[DE] = max3((cpu.pl[FE]+1),` `  opd1, opd2);`
	102	105	103	110	115	`if (cpu.pl[DE] < cpu.pl[EX])` `  cpu.pl[DE] = cpu.pl[EX];`
EXECUTE	102	105	103	110	115	`// INITIAL STATE AT EXECUTE`
	102	105	106	110	115	`cpu.pl[EX] = cpu.pl[DE] + 1;` `*dst1  = cpu.pl[EX];`
	102	105	110	110	115	`if (cpu.pl[EX] < cpu.pl[ME])` `  cpu.pl[EX] = cpu.pl[ME];`
MEMORY	102	105	110	110	115	`// INITIAL STATE AT MEMORY`
	102	105	110	111	115	`cpu.pl[ME] = cpu.pl[EX] + 0;` `*dst2  = cpu.pl[ME];`
	102	105	110	115	115	`if (cpu.pl[ME] < cpu.pl[WB])` `  cpu.pl[ME] = cpu.pl[WB];`
WRITEBACK	102	105	110	115	115	`// INITIAL STATE AT WRITEBACK`
	102	105	110	115	116	`cpu.pl[WB] = cpu.pl[ME] + 1;`
	102	105	110	115	116	`// FINAL PIPELINE STATE`
	Per Pipeline Stage Cycle Count					

Fig. 4. ENCORE 5-Stage Pipeline model example using final instruction from ARCompact™ basic block depicted in Figure 2. It demonstrates the reconstruction of microarchitectural pipeline state *after* the instruction has been executed. Bold red numbers denote changes to cycle-counts for the respective pipeline stages, bold green numbers denote already committed cycle-counts.

3.2 Instruction Operand Dependencies and Side Effects

In order to determine when an instruction is ready to leave the decode stage it is necessary to know when operands become available. For instructions that have side-effects (i.e. modify the contents of a register) we need to remember when the side-effect will become visible. The `avail[GPRS]` array (see label ⑦ in Figure 2) encodes this information for each operand.

When emitting calls to microarchitectural update functions our JIT DBT engine passes source operand availability times and pointers to destination operand availability locations determined during dependency analysis as parameters (see label ③ in Figure 2). This information is subsequently used to compute when an instruction can leave the decode stage (see label ③ in Figure 3) and to record when side-effects become visible in the execute and memory stage (see labels ④ and ⑤ in Figure 3). Because not all instructions modify general purpose registers or have two source operands, there exist several highly optimised versions of microarchitectural state update functions, and the function outlined in Figure 3 demonstrates only one of several possible variants.

3.3 Control Flow and Branch Prediction

When dealing with control flow operations (e.g. jump, branch, branch on compare) special care must be taken to account for various types of penalties and speculative execution. The ARCompact™ ISA allows for delay slot instructions and the ENCORE processor and ARCSIM simulator support various static and dynamic branch prediction schemes.

The code highlighted by label ④ in Figure 2 demonstrates how a branch penalty is applied for a mis-predicted branch. The pipeline penalty depends on the pipeline stage when the branch outcome and target address are known (see target address availability for BCC/JCC and BRCC/BBIT control flow instructions in Figure 3) and the availability of a delay slot instruction. One also must take care of speculatively fetched and executed instructions in case of a mis-predicted branch.

3.4 Cache and Memory Model

Because cache misses and off-chip memory access latencies significantly contribute towards the final cycle count, ARCSIM maintains a 100% accurate cache and memory model. In its default configuration the ENCORE processor implements a pseudo-random block replacement policy where the content of a shift register is used in order to determine a *victim* block for eviction. The rotation of the shift register must be triggered at the same time and by the same events as in hardware, requiring a faithful microarchitectural model.

Because the ARCompact™ ISA specifies very flexible and powerful `load/store` operations, memory access simulation is a critical aspect of high-speed full system simulations. [27] describes in more detail how memory access simulation is implemented in ARCSIM so that accurate modelling of target memory semantics is preserved whilst simulating `load` and `store` instructions at the highest possible rate.

Vendor & Model	HP™ COMPAQ™ dc7900 SFF
Number CPUs	1 (dual-core)
Processor Type	Intel© Core™ 2 Duo processor E8400
Clock Frequency	3 GHz
L1-Cache	32K Instruction/Data caches
L2-Cache	6 MB
FSB Frequency	1333 MHz

Table 1. Simulation Host Configuration.

4 Empirical Evaluation

We have extensively evaluated our cycle-accurate JIT DBT performance modelling approach and in this section we describe our experimental setup and methodology before we present and discuss our results.

4.1 Experimental Setup and Methodology

We have evaluated our cycle-accurate JIT DBT simulation approach using the BIOPERF benchmark suite that comprises a comprehensive set of computationally-intensive life science applications [5]. We also used the industry standard EEMBC 1.1, and COREMARK [36] embedded benchmark suites comprising applications from the automotive, consumer, networking, office, and telecom domains.

All codes have been built with the ARC port of the GCC 4.2.1 compiler with full optimisation enabled (i.e. `-O3 -mA7`). Each benchmark has been simulated in a standalone manner, without an underlying operating system, to isolate benchmark behaviour from background interrupts and virtual memory exceptions. Such system-related effects are measured by including a Linux full-system simulation in the benchmarks.

The BIOPERF benchmarks were run with "class-A" input data-sets available from the BIOPERF web site. The EEMBC 1.1 and COREMARK benchmarks were configured using large iteration counts to execute at least 10^9 instructions. All benchmarks were simulated until completion. The Linux benchmark consisted of simulating the boot-up and shut-down sequence of a Linux kernel configured to run on a typical embedded ARC700 system with two interrupting timers, a console UART, and a paged virtual memory system.

Our main interest has been on simulation *speed*, therefore we have measured the maximum possible simulation speed in MIPS using various simulation modes (FPGA speed *vs.* cycle-accurate interpretive mode *vs.* cycle-accurate JIT DBT mode - see Figures 5, 6, 7 and 8). Table 2 lists the configuration details of our simulator and target processor. All measurements were performed on a X86 desktop computer detailed in Table 1 under conditions of low system load. When comparing ARCSIM simulation speeds to FPGA implementations shown in Figures 5, 6, 7 and 8, we used a XILINX VIRTEX5 XC5 VFX70T (speed grade 1) FPGA clocked at 50 MHz.

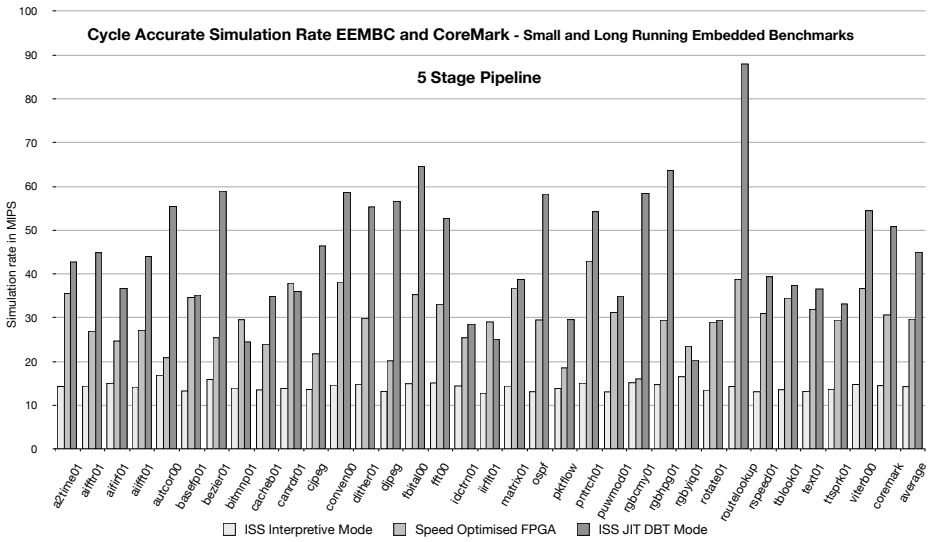

Fig. 5. 5-Stage Pipeline - Simulation rate (in MIPS) using EEMBC and COREMARK benchmarks comparing (a) ISS interpretive cycle-accurate simulation mode, (b) speed-optimised FPGA implementation, and (c) our novel ISS JIT DBT cycle-accurate simulation mode.

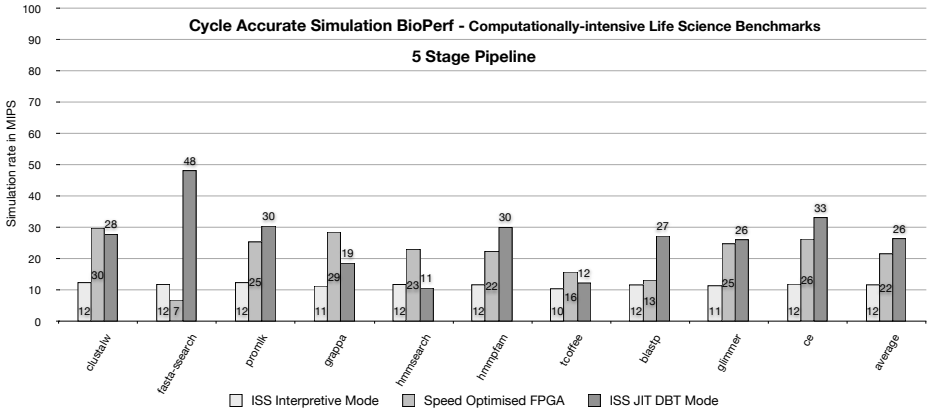

Fig. 6. 5-Stage Pipeline - Simulation rate (in MIPS) using the BIOPERF benchmarks comparing (a) ISS interpretive cycle-accurate simulation mode, (b) speed-optimised FPGA implementation, and (c) our novel ISS JIT DBT cycle-accurate simulation mode.

Processor Microarchitecture	ENCORE
Pipeline	5-Stage and 7-Stage Interlocked
Execution Order	In-Order
Branch Prediction	Yes
ISA	ARCompact™
Register Set	32 baseline registers
Instruction Set Extensions	None
Floating-Point	Hardware
Memory System	
L1-Cache	
Instruction	32k/4-way associative
Data	32k/4-way associative
Replacement Policy	Pseudo-random
L2-Cache	None
Bus Width/Latency/Clock Divisor	32-bit/16 cycles/2
Instruction Set Simulator	ARCSIM
Simulator	Full-system, cycle-accurate
JIT Compiler	LLVM 2.7
I/O & System Calls	Emulated

Table 2. Configuration and setup of simulated target microarchitectures and the ISS. FPGA and ASIP implementations of the outlined microarchitectures were used for verification.

4.2 Simulation Speed

We initially discuss the simulation speed-up achieved by our novel cycle-accurate JIT DBT simulation mode compared to a verified cycle-accurate interpretive simulation mode for a 5-stage processor pipeline variant as this has been the primary motivation of our work. Finally, we also outline results for a different pipeline variant, namely the 7-stage pipeline version of the ENCORE. A summary of our results is shown in Figures 5, 6, 7, and 8.

For EEMBC and COREMARK benchmarks (Figure 5) our proposed cycle-accurate JIT DBT simulation mode for the 5-stage pipeline variant is more than *three* times faster on average (45 MIPS) than the verified cycle-accurate interpretive mode (14 MIPS). It even outperforms a speed-optimised FPGA implementation of the ENCORE processor (30 MIPS) clocked at 50 MHz. For some benchmarks (e.g. `autcor00`, `bezier01`, `cjpeg`, `djpeg`, `rgbcmy01`, `rgbhpg01`, `routelookup`) our new cycle-accurate JIT DBT mode is more than *twice* as fast as the speed-optimised FPGA implementation. This can be explained by the fact that those benchmarks contain sequences of instructions that map particularly well onto the simulation host ISA. Furthermore, frequently executed blocks in these benchmarks contain instructions with fewer dependencies resulting in the generation and execution of simpler microarchitectural state update functions.

Our cycle-accurate JIT DBT simulation achieves an average simulation rate of 26 MIPS for the computationally-intensive life science application programs from the BIOP-

ERF benchmark suite (Figure 6), again outperforming the previously outlined speed-optimised FPGA implementation (22 MIPS). For the `fasta-ssearch` benchmark our JIT DBT is more than 6 times faster than the speed-optimised FPGA which is due to a relatively high cycles per instruction (CPI) metric of 7. For the `hmmsearch` benchmark our JIT DBT cycle accurate simulation is slightly slower than interpretive cycle accurate simulation. This is entirely due to the shorter runtime and abundance of application hotspots keeping the JIT DBT engine very busy, resulting in a slowdown due to JIT compilation overheads.

For EEMBC and COREMARK benchmarks our cycle-accurate JIT DBT simulation mode for the 7-stage pipeline variant (Figure 7) is more than *twice* as fast on average (33 MIPS) than the verified cycle-accurate interpretive mode (13 MIPS). Again it outperforms the speed-optimised FPGA implementation of the 7-stage ENCORE processor variant (28 MIPS) clocked at 50 MHz. For some benchmarks (e.g. `autcor00`, `djpeg`, `rgbcmy01`) our new cycle-accurate JIT DBT mode is almost *twice* as fast as the speed-optimised FPGA implementation. Average BIOPERF benchmark simulation rate figures for the 7-stage pipeline (Figure 8) demonstrate that our cycle-accurate JIT DBT (22 MIPS) once more outperforms a speed-optimised FPGA implementation (21 MIPS) and is *twice* as fast as cycle-accurate interpretive simulation (11 MIPS).

For the introductory sample application performing AAC decoding and playback of Mozart's *Requiem* outlined in Section 1, our cycle-accurate JIT DBT mode is capable of simulating at a sustained rate of 31 MIPS (7-stage pipeline) and 36 MIPS (5-stage pipeline), enabling real-time simulation. For the boot-up and shutdown sequence of a Linux kernel our fast cycle-accurate JIT DBT simulation mode achieves 12 MIPS for both pipeline variants resulting in a highly responsive interactive environment. These examples clearly demonstrate that ARCSIM is capable of simulating system-related effects such as interrupts and virtual memory exceptions efficiently and still provide full microarchitectural observability.

Our *profiling* simulation mode is orthogonal to all of the above simulation modes. Note that for all performance results full profiling was enabled (including dynamic instruction execution profiling, per instruction latency distributions, detailed cache statistics, executed delay slot instructions, as well as various branch predictor statistics).

5 Related Work

Previous work on high-speed instruction set simulation has tended to focus on compiled and hybrid mode simulators. Whilst an interpretive simulator spends most of its time repeatedly fetching and decoding target instructions, a compiled simulator fetches and decodes each instruction once, spending most of its time performing the operations.

5.1 Fast Instruction Set Simulation

A statically-compiled simulator [18] which employed in-line macro expansion was shown to run up to three times faster than an interpretive simulator. Target code is statically translated to host machine code which is then executed directly within a switch statement.

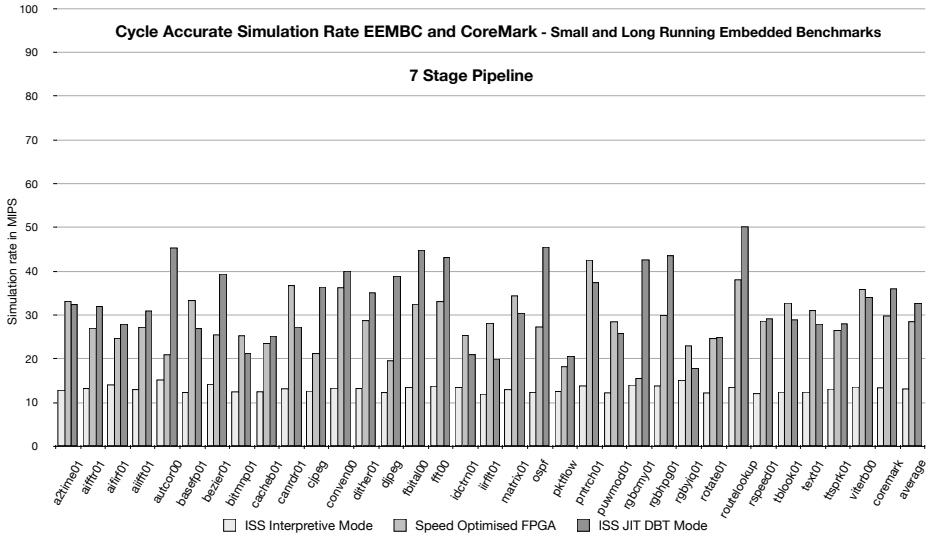

Fig. 7. 7-Stage Pipeline - Simulation rate (in MIPS) using EEMBC and COREMARK benchmarks comparing (a) ISS interpretive cycle-accurate simulation mode, (b) speed-optimised FPGA implementation, and (c) our novel ISS JIT DBT cycle-accurate simulation mode.

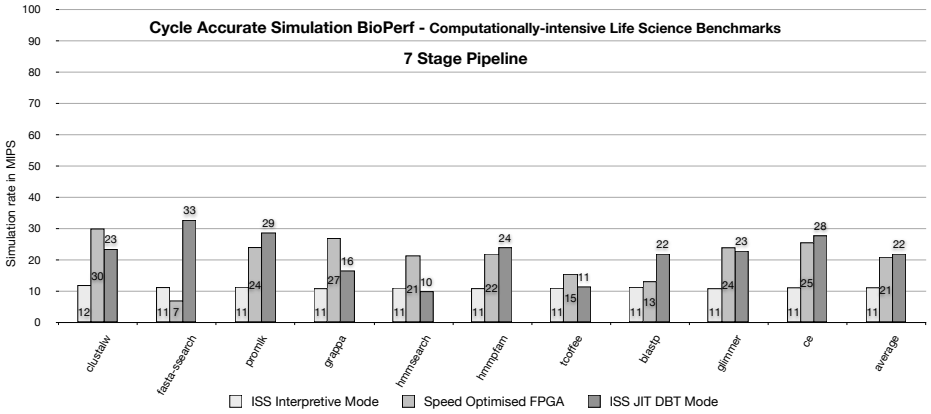

Fig. 8. 7-Stage Pipeline - Simulation rate (in MIPS) using the BIOPERF benchmarks comparing (a) ISS interpretive cycle-accurate simulation mode, (b) speed-optimised FPGA implementation, and (c) our novel ISS JIT DBT cycle-accurate simulation mode.

Dynamic translation techniques are used to overcome the lack of flexibility inherent in statically-compiled simulators. The MIMIC simulator [17] simulates IBM SYSTEM/370 instructions on the IBM RT PC and translates groups of target basic blocks into host instructions. SHADE [9] and EMBRA [28] use DBT with translation caching techniques in order to increase simulation speeds. The Ultra-fast Instruction Set Simulator [30] improves the performance of statically-compiled simulation by using low-level binary translation techniques to take full advantage of the host architecture.

Just-In-Time Cache Compiled Simulation (JIT-CCS) [21] executes and caches pre-compiled instruction-operation functions for each function fetched. The Instruction Set Compiled Simulation (IC-CS) simulator [25] was designed to be a high performance and flexible functional simulator. To achieve this the time-consuming instruction decode process is performed during the compile stage, whilst interpretation is enabled at simulation time. The SIMICS [25] full system simulator translates the target machine-code instructions into an intermediate format before interpretation. During simulation the intermediate instructions are processed by the interpreter which calls the corresponding service routines. QEMU [3] is a fast simulator which uses an original dynamic translator. Each target instruction is divided into a simple sequence of micro-operation, the set of micro-operations having been pre-compiled offline into an object file. During simulation the code generator accesses the object file and concatenates micro-operations to form a host function that emulates the target instructions within a block. More recent approaches to JIT DBT ISS are presented in [24,27,6,15,7]. Apart from different target platforms these approaches differ in the granularity of translation units (basic blocks vs pages or CFG regions) and their JIT code generation target language (ANSI-C vs LLVM IR).

The commercial simulator XISS simulator [35] employs JIT DBT technology and targets the same ARCompact™ ISA that has been used in this paper. It achieves simulation speeds of 200+ MIPS. In contrast, ARCSIM operates at 500+ MIPS [7] in functional simulation mode.

5.2 Performance Modelling in Fast Instruction Set Simulators

A dynamic binary translation approach to architectural simulation has been introduced in [8]. The POWERPC ISA is dynamically mapped onto PISA in order to take advantage of the underlying SIMPLESCALAR [31] timing model. While this approach enables hardware design space exploration it does not provide a faithful performance model for any actual POWERPC implementation.

Most relevant to our work is the performance estimation approach in the HYSIM hybrid simulation environment [11,16,12,13]. HYSIM merges native host execution with detailed ISS. For this, an application is partitioned and operation cost annotations are introduced to a low-level intermediate representation (IR). HYSIM "imitates" the operation of an optimising compiler and applies generic code transformations that are expected to be applied in the actual compiler targeting the simulation platform. Furthermore, calls to stub functions are inserted in the code that handle accesses to data managed in the ISS where also the cache model is located. We believe there are a number of short-comings in this approach: First, no executable for the target platform is ever generated and, hence, the simulated code is only an approximation of what the actual target

compiler would generate. Second, no detailed pipeline model is maintained. Hence, cost annotations do not reflect actual instruction latencies and dependencies between instructions, but assume fixed average instruction latencies. Even for relatively simple, non-superscalar processors this assumption does not hold. Furthermore, HySim has only been evaluated against an Iss that does not implement a detailed pipeline model. Hence, accuracy figures reported in e.g. [12] only refer to how close performance estimates come to those obtained by this Iss, but it is unclear if these figures accurately reflect the actual target platform. Finally, only a very few benchmarks have been evaluated. A similar hybrid approach targeting software energy estimation has been proposed earlier in [19,20].

Statistical performance estimation methodologies such as SimPoint and Smarts have been proposed in [14,29]. The approaches are potentially very fast, but require preprocessing (SimPoint) of an application and do not accurately model the microarchitecture (Smarts, SimPoint). Unlike our accurate pipeline modelling this introduces a statistical error that cannot be entirely avoided.

Machine learning based performance models have been proposed in [2,4,22] and, more recently, more mature approaches have been presented in [10,23]. After initial training these performance estimation methodologies can achieve very high simulation rates that are only limited by the speed of faster, functional simulators. Similar to Smarts and SimPoint, however, these approaches suffer from inherent statistical errors and the reliable detection of statistical outliers is still an unsolved problem.

6 Summary and Conclusions

We have demonstrated that our approach to cycle-accurate Iss easily surpasses speed-optimised Fpga implementations whilst providing detailed architectural and microarchitectural profiling feedback and statistics. Our main contribution is a simple yet powerful software pipeline model in conjunction with an instruction operand dependency and side-effect analysis pass integrated into a Jit Dbt Iss enabling ultra-fast simulation speeds without compromising microarchitectural observability. Our cycle-accurate microarchitectural modelling approach is portable and independent of the implementation of a functional Iss. More importantly, it is capable of capturing even complex interlocked processor pipelines. Because our novel pipeline modelling approach is microarchitecture adaptable and decouples the performance model in the Iss from functional simulation it can be automatically generated from Adl specifications.

In future work we plan to improve and optimise Jit generated code that performs microarchitectural performance model updates and show that fast cycle-accurate multicore simulation is feasible with our approach.

References

1. David August, Jonathan Chang, Sylvain Girbal, Daniel Gracia Perez, Gilles Mouchard, David Penry, Olivier Temam, and Neil Vachharajani. UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development. IEEE Computer Architecture Letters, 20 Aug (2007).
2. J. R. Bammi, E. Harcourt, W. Kruijtzer, L. Lavagno, and M. T. Lazarescu. Software performance estimation strategies in a system-level design tool. In Proceedings of CODES'00, (2000).
3. F. Bellard. QEMU, a fast and portable dynamic translator. Proceedings of the Annual Conference on USENIX Annual Technical Conference. USENIX Association, Berkeley, CA, p. 41, (2005).
4. G. Bontempi and W. Kruijtzer. A data analysis method for software performance prediction. DATE'02: Proceedings of the Conference on Design, Automation and Test in Europe, (2002).
5. D. Bader, Y. Li, T. Li, and V. Sachdeva. BioPerf: A Benchmark Suite to Evaluate High-Performance Computer Architecture on Bioinformatics Applications. In: Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'05), pp. 163–173, 2005.
6. Florian Brandner, Andreas Fellnhofer, Andreas Krall, and David Riegler Fast and Accurate Simulation using the LLVM Compiler Framework. RAPIDO'09: 1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (2009) pp. 1-6.
7. Igor Böhm, Björn Franke and Nigel Topham Cycle-Accurate Performance Modelling in an Ultra-Fast Just-In-Time Dynamic Binary Translation Instruction Set Simulator. In: Proceedings of the International Symposium on Systems, Architectures, Modeling, and Simulation (SAMOS'10), Samos, Greece, (2010)
8. H.W. Cain, K.M. Lepak, and M.H. Lipasti. A dynamic binary translation approach to architectural simulation. SIGARCH Computer Architecture News, Vol. 29, No. 1, March (2001).
9. B. Cmelik, and D. Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp. 128–137, ACM Press, New York, (1994).
10. Björn Franke. Fast cycle-approximate instruction set simulation. SCOPES'08: Proceedings of the 11th international workshop on Software & compilers for embedded systems (2008).
11. Lei Gao, Stefan Kraemer, Rainer Leupers, Gerd Ascheid, Heinrich Meyr. A fast and generic hybrid simulation approach using C virtual machine. CASES'07: Proceedings of the international conference on Compilers, architecture, and synthesis for embedded systems (2007).
12. Lei Gao, Stefan Kraemer, Kingshuk Karuri, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. An Integrated Performance Estimation Approach in a Hybrid Simulation Framework. MOBS'08: Annual Workshop on Modelling, Benchmarking and Simulation (2008).
13. Lei Gao, Kingshuk Karuri, Stefan Kraemer, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Multiprocessor performance estimation using hybrid simulation. DAC'08: Proceedings of the 45th annual Design Automation Conference (2008).
14. G. Hamerly, E. Perelman, J. Lau, and B. Calder. SIMPOINT 3.0: Faster and more flexible program analysis. MOBS'05: Proceedings of Workshop on Modelling, Benchmarking and Simulation, (2005).
15. Daniel Jones and Nigel Topham. High Speed CPU Simulation Using LTU Dynamic Binary Translation. Lecture Notes In Computer Science (2009) vol. 5409.
16. Stefan Kraemer, Lei Gao, Jan Weinstock, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. HySim: a fast simulation framework for embedded software development. CODES+ISSS'07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis (2007).

17. C. May. MIMIC: A Fast System/370 Simulator. SIGPLAN: Papers of the Symposium on Interpreters and Interpretive Techniques, pp. 1–13, ACM Press, Ney York, (1987).
18. C. Mills, S.C. Ahalt, J. Fowler. Compiled Instruction Set Simulation. Software: Practice and Experience, 21(8), pp. 877 – 889, (1991).
19. A. Muttreja, A. Raghunathan, S. Ravi, and N.K. Jha. Hybrid simulation for embedded software energy estimation. DAC'05: Proceedings of the 42nd Annual Conference on Design Automation, pp. 23–26, ACM Press, New York, (2005).
20. A. Muttreja, A. Raghunathan, S. Ravi, and N.K. Jha. Hybrid simulation for energy estimation of embedded software. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, (2007).
21. A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann. A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation. DAC'02: Proceedings of the 39th Conference on Design Automation, pp. 22–27, ACM Press, New York, (2002).
22. M. S. Oyamada, F. Zschornack, and F. R. Wagner. Accurate software performance estimation using domain classification and neural networks. In Proceedings of SBCCI'04, (2004).
23. Daniel Powell and Björn Franke. Using continuous statistical machine learning to enable high-speed performance prediction in hybrid instruction-/cycle-accurate instruction set simulators. CODES+ISSS'09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis, (2009).
24. W. Qin, J. D'Errico, and X. Zhu. A Multiprocessing Approach to Accelerate Retargetable and Portable Dynamic-Compiled Instruction-Set Simulation. CODES-ISSS'06: Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis, pp. 193–198, ACM Press, New York, (2006).
25. M. Reshadi, P. Mishra, and N. Dutt. Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation. Proceedings of the 40th Conference on Design Automation, pp. 758–763, ACM Press, New York, (2003).
26. O. Schliebusch, A. Hoffmann, A. Nohl, G. Braun, and H. Meyr. Architecture Implementation Using the Machine Description Language LISA. ASP-DAC'02: Proceedings of the Asia and South Pacific Design Automation Conference, Washington, DC, USA, (2002).
27. Nigel Topham and Daniel Jones. High Speed CPU Simulation using JIT Binary Translation. MOBS'07: Annual Workshop on Modelling, Benchmarking and Simulation (2007).
28. E. Witchel, and M. Rosenblum. Embra: Fast and Flexibile Machine Simulation. In: Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, pp. 68–79, ACM Press, New York, (1996).
29. R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. ISCA'03: Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA), (2003).
30. J. Zhu, and D.D. Gajski. A Retargetable, Ultra-Fast Instruction Set Simulator. DATE'99: Proceedings of the Conference on Design, Automation and Test in Europe, p. 62, ACM Press, New York, (1999).
31. Doug Burger and Todd Austin. The SimpleScalar tool set, version 2.0. SIGARCH Computer Architecture News (1997) vol. 25 (3).
32. ARCompact™ Instruction Set Architecture. Synopsys Inc. `http://www.synopsys.com/IP/ConfigurableCores/ARCProcessors/`, retrieved 05 November (2010).
33. ENCORE Embedded Processor. `http://groups.inf.ed.ac.uk/pasta/hw_encore.html`, retrieved 05 November 2010.
34. ARCSIM Instruction Set Simulator. `http://groups.inf.ed.ac.uk/pasta/tools_arcsim.html`, retrieved 05 November 2010.
35. XISS Simulator. Synopsys Inc. `http://www.synopsys.com/dw/ipdir.php?ds=sim_xiss`, retrieved 10 February 2010.

36. The Embedded Microprocessor Benchmark Consortium: EEMBC Benchmark Suite. http://www.eembc.org