THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

High Speed CPU Simulation using LTU Dynamic Binary Translation

Citation for published version:
Jones, D & Topham, NP 2009, High Speed CPU Simulation using LTU Dynamic Binary Translation. in *4th International Conference on High Performance Embedded Architectures and Compilers (HiPEAC).* Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 50-64. https://doi.org/10.1007/978-3-540-92990-1_6

Digital Object Identifier (DOI):
10.1007/978-3-540-92990-1_6

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
4th International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)

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

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

Download date: 20. Apr. 2024

High Speed CPU Simulation using JIT Binary Translation

Nigel Topham
npt@inf.ed.ac.uk
Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh

Daniel Jones
daniel.jones@ed.ac.uk
Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh

***Abstract*—Instruction set simulators are indispensable tools for exploring the design-space of innovative processor architectures, for processor verification, and for software development. Traditional interpretive simulators are too slow to cope with the increasing complexity of embedded processors now being deployed in many high performance systems. High speed emulation techniques based on dynamic binary translation have been proposed previously, but thus far we have not seen flexible multi-function full-system simulators capable of acting as golden reference models, software development platforms and design-space exploration tools. This paper presents a target-adaptable full-system simulator which combines the speed of JIT binary translation with the observability of interpreted simulation. We explain the mechanisms it uses to achieve sufficiently high performance to boot and run Linux interactively at speeds exceeding those achievable with FPGA-based RTL emulation of the same processor. We report performance figures from a set of representative embedded benchmarks which range from 187 to 373 MIPS. Our results also indicate that transient simulation speeds can exceed 1,000 MIPS, and we show that a full-system Linux simulation can sustain more than 148 MIPS.**

I. Introduction

Simulators play an important multi-function role in the design of today's high performance processors. They enable application specific (ASIP) design-space exploration, as well as providing an essential tool for software developers. Processor characteristics such as speed and power consumption may be accurately predicted by a simulator, allowing the most efficient design to be adopted for fabrication. Simulators also facilitate the testing of experimental ISAs, and the development and verification of new compilers and applications. Simulation platforms enable concurrent engineering during the design of a new microprocessor, reducing the overall development time. This is especially important for embedded system-on-chip designs, where processors may be extended to support specific applications.

The ever-increasing complexity of modern microprocessors exacerbates the central challenge of CPU simulation: to achieve high-speed simulation whilst retaining absolute modelling accuracy. In addition to modelling complex events such as interrupts and exceptions, simulators need to be highly configurable. Many embedded processors now incorporate memory management hardware to support multi-tasking operating systems. This presents a significant challenge for high speed simulators as virtual memory translation can easily become a bottleneck.

The class of simulators with which we are concerned, are required to provide *accurate* and *observable* modelling of the entire processor state. This is possible to achieve by operating at the register transfer level, but such simulations are very slow. In contrast, a compiled simulation [1] can be many orders of magnitude faster, but does not have the same degree of observability, and can only be used in situations where the application code is known in advance and is available in source form. Programs which require an operating system, which are self-modifying, or which are shrink-wrapped, can not benefit from compiled simulation.

Dynamic JIT binary translation combines interpretive and compiled simulation techniques, maintaining both flexibility and high simulation speeds. However, achieving accurate state observability remains in tension with high performance.

This paper presents a high-speed simulator which provides both interpretive and JIT binary translation, whilst also providing full observability of all target state changes. The simulator also captures all architecturally-visible CPU state changes as instructions commit, in order to support high-speed hardware-software co-verification.

Section II presents an overview of the simulator and its modes of operation. Section III describes the novel mechanism which supports high-speed memory access operations in the presence of virtual address translation and memory-mapped I/O. Sections IV, V and VI respectively explain the interpretive mode of simulation, the method of profiling which takes place during interpretation, and the dynamic JIT translation scheme. Performance results are presented in section VII, followed by a brief discussion of related work in section IX and concluding remarks in section X.

II. Simulator Overview

The simulator presented in this paper is target-adaptable, with the initial target being the ARC 700™ processor which implements the ARCompact™ instruction set architecture [2]. 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. In contrast

with other high-speed functional simulators, we maintain a precise view of the target processor state. This allows the simulator to be used as a software development platform as well as a tool for functional verification of customised processors derived from the ARC 700 baseline processor.

There are two modes of operation: an interpretive mode which provides precise observability after each instruction; and a just-in-time (JIT) binary translation mode which provides similarly precise observability between successive basic blocks.

In common with other conventional interpretive simulator, such as SimpleScalar [3], the interpretive mode repeatedly fetches, decodes and interprets successive instructions in the execution path. Registers, memory and the context of I/O devices are updated as instructions commit, thereby maintaining a precise view of the target system.

Functions which update simulation statistics, and which model micro-architectural features of the processor, may be enabled at appropriate points in either mode of operation. These typically count program events, model caches, or even predict power consumption. Interpretive simulators typically execute approximately 100 host instructions per target instruction, although this may rise significantly depending on the accuracy of timing or power modelling.

The JIT binary translation mode combines the speed of compiled simulation with the flexibility of interpretive simulation, allowing even shrink-wrapped binaries to be simulated at high speed. When running in this mode the simulator initially operates interpretively, discovering and profiling the basic blocks as they are encountered. The simulator periodically examines the execution profile looking for frequently executed blocks, which are thus identified for binary translation. Binary translation in this context is the process of translating target basic blocks to semantically-equivalent native code which can then be run on the host machine [4]. The locality present in most programs means that a few basic blocks will be executed many times. A basic block which has been previously translated will then be simulated with much greater speed by simply executing the translation.

In this simulator, the underlying system architecture for handling memory access, I/O, interrupts, and exceptions is the same for both the interpreted and translated code. This allows the simulator to switch between modes at the granularity of a single target instruction. At any time, a translated block can be terminated and simulation restarted at the current program counter location. This enables translated blocks to raise exceptions part-way through, after which the remaining instructions in the block will be interpreted.

III. Simulating Memory Accesses

Memory access simulation is a critical aspect of high speed full-system simulators. Our goal, when simulating memory accesses, is to achieve accurate modelling of the target memory system semantics whilst simulating `load` and `store` instructions at the highest possible rate.

The simulator simulates memory accesses based on target processor addresses, which will be virtual addresses when the MMU is enabled. The requirement for accurate event modelling means that each memory-related exception that would occur on a real target must also occur in precisely the same way and at the same point in the simulated program. To achieve a useable simulation speed, memoization techniques are employed to avoid simulating the full MMU semantics when we can deduce the outcome of a translation without actually interrogating the simulated TLBs.

The functional requirements for memory access simulation can be summarized as:

1) Implementing a mapping from target virtual addresses to host virtual addresses, via the target physical address space.
2) Detecting all forms of memory access exceptions, including: misalignment, access violation and TLB miss.
3) Providing visibility of the outcome of each `load` or `store` instruction upon completion of the instruction.
4) Minimizing the number of host instructions required to implement each target `load` or `store` instruction.

The logical process of translating a target virtual address is illustrated in figure 1a. The TLB contents are used to translate from target virtual address to target physical address, possibly raising a TLB-related exception. If no exception occurs, the corresponding location of the physical page in host memory must be determined. As target memory may be sparsely populated, a Standard Template Library `map<>` container is used to hold all such mappings.

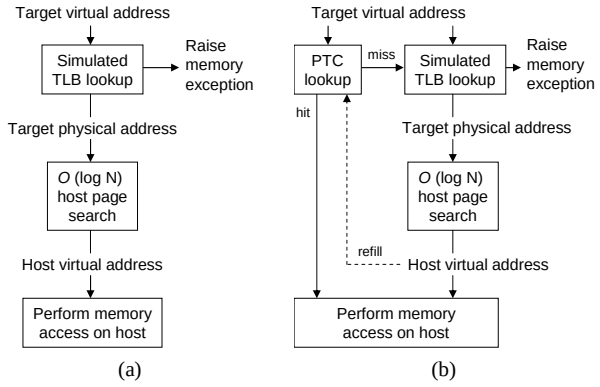


Fig. 1. The process of translating target memory references: fig.(a) shows the sequence of translation steps and look-ups required in principle; fig.(b) shows the memoized translation process.

In practice, these address mappings are memoized by introducing three Page Translation Cache (PTC) structures, one each for Read, Write and Fetch accesses, as shown in figure 1b. Each PTC is a direct-mapped software cache, indexed by low-order target virtual address bits. Each valid entry memoizes a target page if, and only if: the following:

- The associated target virtual address is currently mapped by one of the target's TLB entries.
- The current process has sufficient privilege to access the page.

- When present in the Read PTC, the page is Read Permitted. Similarly, when present in the Write PTC or the Fetch PTC, the page must be Write or Execute permitted respectively.
- The physical page is implemented as ordinary external memory with no side-effects on Read or Write.

By having three separate PTCs, the simulator can trap write accesses to read-only pages whilst providing full-speed read or execute access to those pages. The PTC can also be used to trap self-modifying code by enforcing a rule that any physical page, for which there is an entry in the Fetch PTC, may not be also present in the Write PTC, and *vice versa*. A write access to a page that is present in the Fetch PTC will be *trapped* by a Write PTC miss, resulting in the flushing of all dynamic translations or cached instruction decodings for that page. A subsequent Fetch access to that page will be trapped by a Fetch PTC miss, which will remove its entry from the Write PTC. This takes care of virtual aliasing between processes which may have different permissions to access the same physical page (common in Linux, for example).

Each PTC tag has several spare bits which can be used to flag special properties of the associated page. If present, such property bits result in a PTC miss on *every* access. However, the miss handler quickly determines the presence of the property and processes it as necessary. For example, there is a property bit for memory regions with side-effects on Read or Write. The housekeeping structure for such a page contains a pointer to the Read and Write functions, which can be provided by an external user-defined library. This enables fast and efficient simulation of memory-mapped I/O devices. The data structures to support fast address translation are illustrated in figure 2.

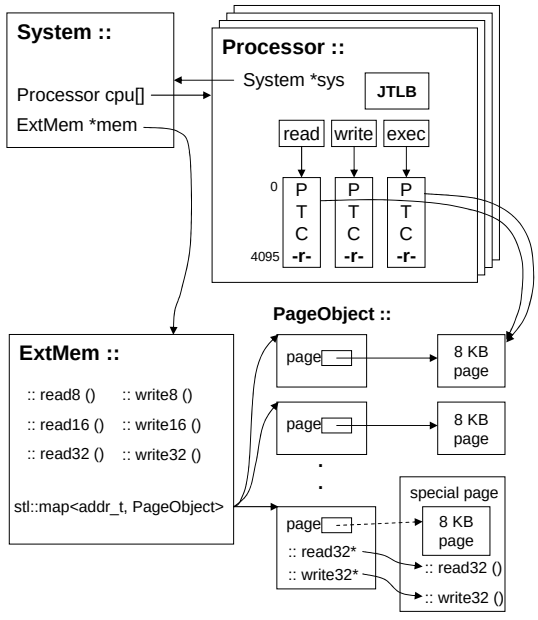


Fig. 2. Address translation structures

A read access which hits in the PTC executes at most

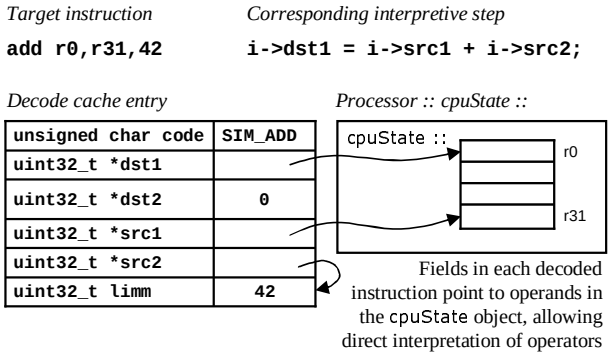


Fig. 3. Decode cache illustration and example `add` instruction.

16 native host instructions. Similarly, a memoized write hit requires 18 native instructions. PTC misses take longer to process, but occur with much lower frequency. Effectively, the PTCs bypass MMU translation as well as bypassing the process of locating the actual data page in the host.

IV. Interpretive Mode

The simulator can operate in an interpretive mode which incorporates several features to speed up simulation. The decoding of instructions through the use of `switch` statements is performed in an order such that the most frequently executed instructions are decoded quickest. The simulator incorporates a Decode Cache (DCC) which caches individual decoded instructions. Although the decode process is comparatively time consuming, the high hit rates in the DCC amortize the decode time over many executions of the same instruction. The program counter indexes the DCC when interpreting instructions. On a hit the the fields of the selected decode cache entry point directly to the decoded instruction operands in the processor state structures. On a miss the instruction must first be fetched and then decoded. The decoded instruction is then added to the DCC, after which it is interpreted as above. Figure 3 shows a typical DCC entry for an `add` instruction.

V. Block Profiling

A key feature of the simulator is its ability to translate target instructions into host instructions dynamically. One might reasonably ask why we do not translate target programs statically, and in advance of execution. The answer is threefold:

1) It is not possible in general to discover the regions of the text section that are guaranteed to contain instructions rather than embedded data. Instead, the simulator *discovers* each basic block when it is first executed.
2) The simulator image would be expanded unecessarily by including translated basic blocks that are not essential for high performance. Instead, we only translate blocks that are considered *hot*, leaving *cool* blocks to be interpreted.
3) Simulator responsiveness would suffer if a large quantity of target code has to be translated before any part of

an application can begin simulating. By translating on-demand we are able to spread out the translation effort, making it less noticeable to the user.

The simulator maintains a Basic Block Map (BBM), containing an entry for every basic block encountered during simulation. Each BBM entry contains the block location, the number of instructions in the block, and the number of times it has been interpreted and executed in translated mode. If the block has been translated, there will also be a pointer to the Translated Block Function (TBF) representing the translated version of the block. This method of block profiling allows the simulator to keep track of the number of instructions executed in each block and hence for the entire simulation.

Simulation time is partitioned into *epochs*, where each epoch is defined as the interval between two successive JIT translations. During each epoch new blocks may be discovered; previously seen but non-translated blocks may be re-interpreted; and translated blocks may be executed directly. Throughout this process the simulator continues to build up a profile of every block executed, regardless of the execution mode. The end-point of an epoch is reached when sufficient basic blocks have been interpreted. This definition of an epoch ensures that translations are only made when there is something worth translating.

VI. JIT Translation

When JIT translation is enabled, instead of fetching and decoding the next instruction, as occurs in a purely interpretive mode, the simulator checks to see whether the current PC is present in a fast, direct-mapped, Block Translation Cache (BTC), using the sequence of steps illustrated in figure 4. The BTC, which is indexed by basic block entry points, contains the address of each corresponding TBF, allowing the simulator to locate most translations very quickly.

If the PC for the next basic block hits in the BTC, the function pointed to by its TBF will be called, thereby emulating the block directly. In addition to performing all of the operations within the basic block, all state information for the simulated processor, including PC value, is updated prior to exit from the TBF.

If the next PC misses in the BTC, it is looked for in the BBM. If present, and if a translation for the next block exists, the corresponding TBF is obtained from the BBM and loaded into the BTC.

If the next block is not present in the BBM we know this is the first execution of the block. An entry is therefore created in the BBM and a cached entry is created in the Epoch Block Cache (EBC) in order to record all blocks that miss in the BTC and BBM during the current epoch. This is always a subset of the BBM. Each EBC entry also points to its parent entry in the BBM, where the TBF is maintained for each basic block that has been previously translated. Hence, if a basic block has been translated but misses in the BTC, it will be found in the EBC or the BBM unless the entry has been subsequently nullified (*e.g.* due to self-modifying code or page reuse).

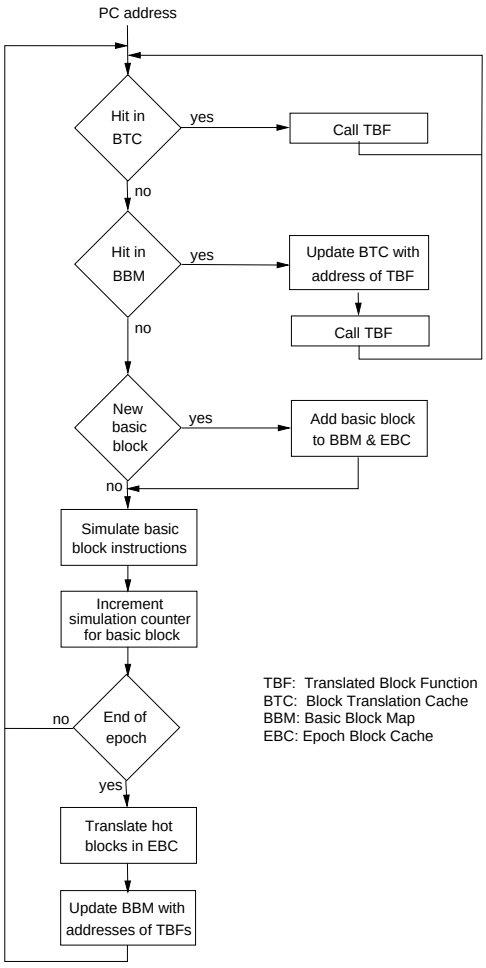


Fig. 4. Flow Chart showing Algorithm for JIT Binary Translation.

At the end of each epoch, all blocks in the EBC are scanned. These represent blocks that have been executed at least once during the last epoch, and are therefore inspected to see if they are hot enough to translate. A simple heuristic based on the number of instructions per blocks and block execution frequency is used to select hot blocks. Hot blocks are then translated in batches comprising blocks from the same physical page of target memory. Grouping blocks in this way enables easy elimination of the entire batch if the target page is overwritten by self-modifying code or (more likely) by the reuse of that physical page for a different process.

When each batch of blocks has been identified for translation, the translations are created and compiled into a shared library which is then loaded using a dynamic linker. Finally, the BBM is updated with the TBF address of each newly-translated basic block. The next time one of the recently translated basic blocks needs simulating, it will hit in the BBM, an entry will be added to the BTC and its TBF called.

A. Example translation

Figure 5 shows a basic block actually identified as hot during Linux boot-up. This is a single-block loop terminating in a branch with a delay-slot.

```
80007214 <dcache_init>:
:
800072ce:  2f23c200          lsr    r3, r3
800072d2:  40224200          add    r2, r2, 1
800072d6:  4c230080          cmp    r3, 0
800072da:  f607e2ff          bnz.d  800072ce
800072de:  0a248000          mov    r4, r2
```

Fig. 5. Example hot basic block from the Linux kernel

Figure 6 shows the translated sequence for this hot block, which comprises 18 native x86 instructions plus a further 5 instructions for the function entry and exit sequence. Three of the `cmp` status outputs are not needed by the `bne.d` instruction, which only uses the Zero condition. However, they cannot be eliminated because complete observability of all state modifications is a functional requirement for simulators such as this one, which can be used as a debugger target. At any observation point the processor state should be indistinguishable from the state at an equivalent observation point in a real processor.

```
L_800072ce:
   pushl   %ebp
   movl    %esp, %ebp
   movl    8(%ebp), %edx
#--------------------------- lsr r3,r3 ---
   movl    20(%edx), %ecx
   shrl    %ecx
   movl    %ecx, 20(%edx)
#--------------------------- add r2,r2,1 -
   incl    16(%edx)
#--------------------------- cmp r3,0 ----
   xorl    %eax, %eax
   cmpl    %eax, %ecx      # r3 is in %ecx
   seto    268(%edx)
   setc    267(%edx)
   sets    266(%edx)
   setz    265(%edx)
#--------------------------- bne.d -10 ---
   movb    265(%edx), %al
   cmpb    $1, %al
   sbbl    %ecx, %ecx
   andl    $-20, %ecx
   subl    $2147454238, %ecx
   movl    %ecx, (%edx)
#--------------------------- mov  r4,r2 --
   movl    16(%edx), %eax
   movl    %eax, 24(%edx)
# ---------------------------------------
   leave
   ret
```

Fig. 6. Translation of example block

B. Analysis of JIT behaviour

Figures 7a and 7b present two related views on the distribution of translated blocks within the Linux kernel after boot up. Figure 7a shows blocks sorted by execution frequency, whereas figure 7b shows them sorted by program address. It is clear that dominant blocks are translated, whereas less frequently executed blocks are not.

Figure 7c shows an instruction profile for both Linux and one of the larger embedded benchmarks, `mpeg2enc`. From this we see that a relatively small number of distinct op-codes account for the majority of the simulated instructions. Using this knowledge the simulator is able to special-case the most frequent variants of multi-variant instructions. For example, `load` instructions have several addressing modes, but only one mode (the simplest) occurs with high frequency.

VII. Performance Measurements

The characteristics of the simulator have been measured extensively to assess its performance across a varied workload. In this section we present a range of performance measurements and explore their implications. All measurements were performed on the server detailed in table I under conditions of low system load.

TABLE I
Simulation host configuration

Parameter	Value
Vendor & model	Dell™ PowerEdge™ 2960
Number CPUs	4 (2 × dual-core)
Processor	Intel® Xeon® 5160
Clock frequency	2992 MHz
L1 caches	32KB I/D caches
L2 cache	4 MB per dual-core CPU
FSB frequency	1333 MHz

Simulator performance depends not only on the speed at which individual target instructions can be simulated on the host, but also on the memory locality characteristics of the application. Our initial benchmarks therefore attempt to isolate core CPU simulation speed from performance degradations due to limitations of the host memory system.

We have observed that most non-memory target instructions translate into sequences of 1 to 5 host instructions. In contrast, the typical path through the translation of a `load` or `store` instruction is 16 and 18 host instructions respectively. For this reason we present micro-benchmarks to illustrate separately the peak simulation speeds of non-memory instructions and memory-referencing instructions.

A. Performance of non-memory instructions

To test the performance of non-memory instructions we constructed a synthetic test comprising a typical sequence of simple arithmetic operations including `add`, `sub`, `cmp`, shift and logical operation, terminating with a branch and its delay slot. The number of instructions in the block was varied between 5 and 25 to isolate the overhead of executing small blocks. The resulting execution speeds versus block size, are shown in table II.

TABLE II
Simulation rate of ALU and Memory operations

Instructions per block	ALU-op MIPS		Mem-op MIPS	
	non-JIT	JIT-enabled	non-JIT	JIT-enabled
5	26	295	28	295
10	27	505	27	357
15	28	711	28	421
20	26	880	26	450
25	29	1,028	29	466

With an unreasonably large block size of 100 simulated instructions, the JIT-translated simulation rate reaches 1,560

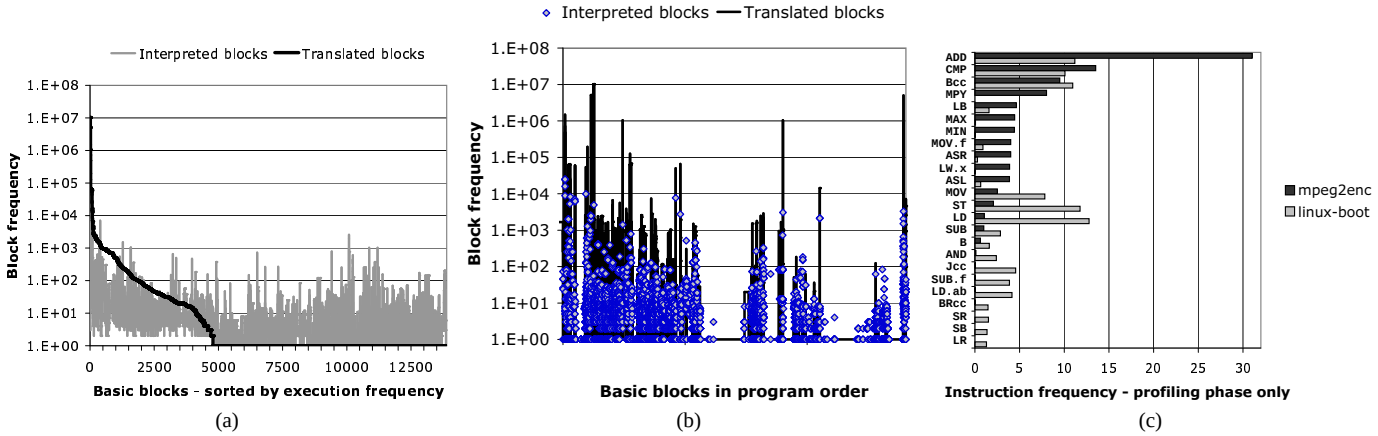


Fig. 7. Execution profiles: charts (a) and (b) shows frequency profiles for all translated and interpreted blocks executed during Linux boot-up, sorted by execution frequency and program address respectively. Chart (c) shows the profiled execution frequencies of all instructions contributing more than 1% to the instruction profiles of Linux and the `mpeg2enc` benchmark.

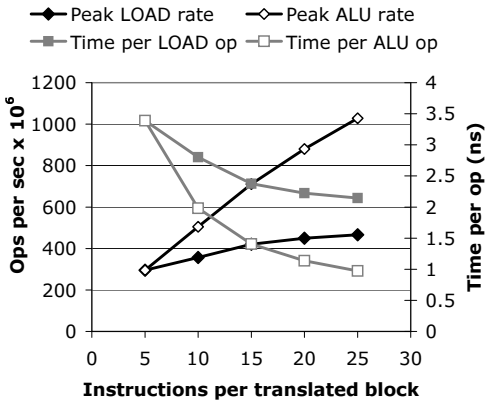


Fig. 8. Peak simulation speeds for ALU ops and memory-access ops.

million instructions per second. In practice such speeds will not be sustained on real programs, although this measurement gives a useful indication of the transient peak simulation rate.

B. *Performance of* `load` *and* `store` *instructions*

To test the performance of `load` and `store` instructions we constructed a synthetic test comprising a typical sequence of memory referencing instructions placed within a loop. In common with the previous experiment, the number of memory access instructions was varied between 5 and 25. The resulting execution speeds versus block size, are shown in cols. 4 and 5 of table II. Figure 8 illustrates the relationship between block size and peak performance for both non-memory operations and memory-access operations. It also shows that the effective time to simulate a single instruction can be as low as 0.9ns for non-memory operations, and 2.14ns for memory-access operations.

As expected, the simulation of memory referencing instructions is slower than other instructions: each memory access instructions is typically implemented by a sequence of 16 to 18 host instructions. A host processor running at 3 GHz, simulating such instructions at a rate of 466×10^6 per second, achieves an IPC of at least 2.5 during simulation. As the code to simulate each individual memory operation has low ILP, the results in table II indicate there is significant overlap between the instructions generated for each of the target instructions in a single basic block.

C. *Benchmark application performance*

In this section we explore simulator performance on a range of embedded benchmarks drawn from MiBench, Dhrystone 2.1 and MediaBench I. As the simulator is fully-capable of executing Linux, we also include the Linux boot-up phase. The selected benchmarks and their characteristics are detailed in table III, which shows their static `.text` section size, the static average number of target instructions per basic block (measured statically and dynamically), and the number of instructions simulated. All benchmarks were run to completion, except for Linux boot-up which was terminated when the console prompt was reached.

Linux has a relatively large `.text` section, compared with other embedded benchmarks. For this reason it represents an interesting test-case for just-in-time dynamic binary translation, motivating a more detailed analysis of Linux behavior later in this section.

TABLE III
BENCHMARK CHARACTERISTICS

Benchmark	Origin	Text (KB)	Ave. block size		Instruction count $\times 10^6$
			Static	Dyn.	
bitcnts	MiBench	36	6.09	9.09	936
dijkstra	MiBench	39	5.74	5.26	56
dhry21	Dhystone 2.1	34	5.85	5.62	5,280
mpeg2enc	MediaBench I	137	6.43	8.75	13,121
qsort	MiBench	37	5.76	7.34	137
10-queens	Traditional	27	5.94	5.89	244
susan	MiBench	60	5.71	9.47	32
linux-boot	Linux 2.1.14	1,378	5.53	5.78	129

All benchmarks were compiled for the ARC 700™ architecture using `gcc` version 3.4.5 with `-O2` optimisation and linked

against `uClibc`. The simulator itself was compiled for an Intel host architecture using `gcc` version 4.1.1 with level `-O4` optimisation. Each benchmark was simulated in a stand-alone manner, without an underlying operating system, to isolate benchmark behavior from background interrupts and virtual memory exceptions. Such system-related effects are measured by including a Linux system simulation in the benchmarks.

The Dhrystone 2.1 benchmark was pre-compiled to run for 10^7 iterations. The well-known N-queens problem was configured to solve a 10×10 problem 12,800 times in succession. The linux-boot benchmark consisted of simulating the boot-up sequence of a Linux kernel configured to run on a typical embedded ARC 700 system with two interrupting timers, a console UART, and paged virtual memory system with a 512-entry 2-way set-associative joint TLB.

One of the key features of the simulator that helps to deliver high performance is the collection of software caches which yield low average search times for some of the more complex data structures in the simulator. Table IV shows the hit ratios for two of the most important software caches, the Decode Cache (DCC) and the Block Translation Cache (BTC).

TABLE IV
DECODE CACHE AND BLOCK TRANSLATION CACHE PERFORMANCE

Benchmark	Cache hit ratios	
	DCC	BTC
bitcnts	98.42	99.77
dijkstra	98.15	99.54
dhry21	95.14	99.99
mpeg2enc	99.83	99.92
qsort	98.73	99.74
queens	98.66	99.93
susan	97.34	98.94
linux-boot	79.71	99.55

Both the DCC and BTC achieve hit rates above 97% in all cases except one – the DCC when booting Linux. The reason for the lower hit ratio of just under 80% is that many small sections of code are touched infrequently during booting. However, as we see in figure 10b, the majority of Linux kernel code is executed in translated mode rather than interpreted mode, so the high BTC hit ratio of 99.55% has a dominant impact on performance in that case.

The overall simulator performance for the all selected benchmarks is shown in figure 9. When JIT binary translation is enabled, the simulator achieves between 187 and 373 simulated MIPS for the stand-alone embedded benchmarks. Linux booting from a cold-start achieves a lower figure of 119 MIPS. However, when the translations are pre-loaded from a previously-saved run, performance rises to 135 MIPS for Linux boot-up. Figure 9a also shows the speed of executing a sequence of typical shell commands on a simulated Linux system, immediately after booting. This achieves 143 MIPS, contrasting favourably with the speeds of 90–100 MHz typically achievable when emulating the ARC 700 on the ARCangel 4™ FPGA-based development platform.

When JIT translation is disabled, performance is relatively stable across all benchmarks at around 22 million instructions per second, as seen in figure 9b. Figure 9c shows the relative fraction of time spent simulating versus translating, as well as the total number of instructions executed by each benchmark.

Given the lower simulation speed observed for Linux, we now examine the dynamic characteristics of Linux simulation during the boot-up phase. To do this, the simulator was instrumented to report simulation and translation times, as well as the number of instructions executed in each mode, during each simulation epoch. The average epoch length was 0.74 seconds, measured over the duration of Linux booting. Figure 10 shows the time-evolving results from each epoch when simulating Linux.

VIII. EXTENDING THE SIMULATOR DYNAMICALLY

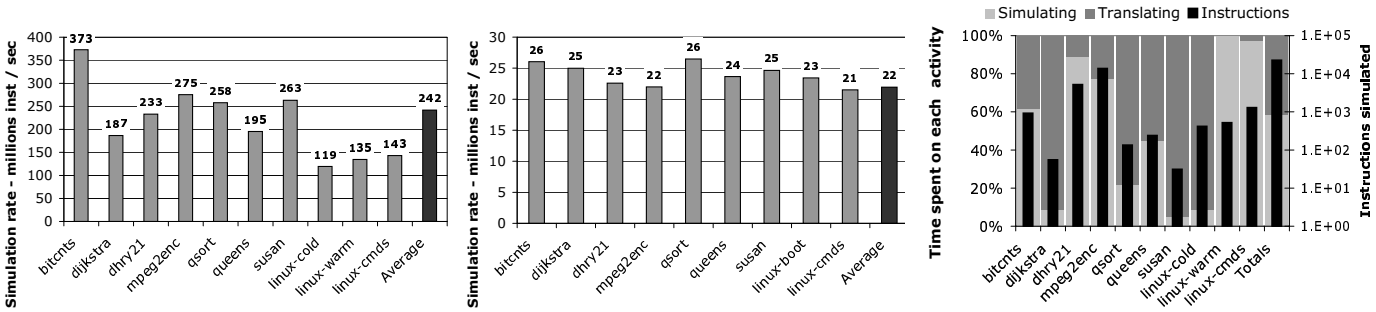
One of the defining features of the ARCompact architecture is its support for user-defined extensions. These may extend the instruction set, the core register file, the set of branch conditions and the auxiliary register set. The simulator provides an API through which extension capabilities can be defined. This is still at an early stage of development but provides, for example, the means to introduce pages of physical memory with side effects. These are registered in the `ExtMem` class, as illustrated in figure 2. Future developments will investigate how the translation mechanism can be extended to handle user-defined instruction set extensions.

IX. RELATED WORK

Prior work on high-speed instruction set simulators has tended to focus on compiled and hybrid mode simulation. A method using in-line macro expansion capabilities is illustrated in [1]. Target code is statically translated to host machine code which is then executed directly. It was demonstrated that a statically-compiled simulator could run up to three times faster than an interpretive simulator.

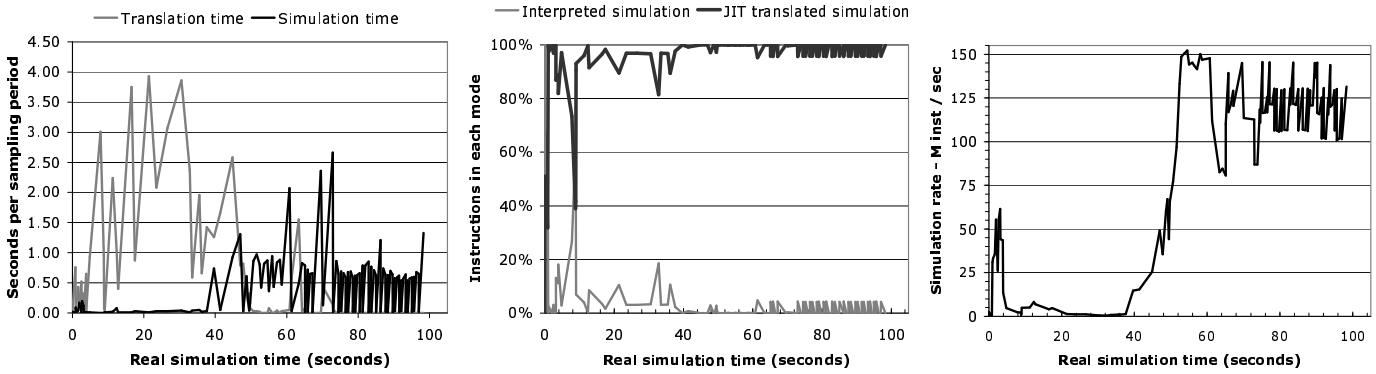
Dynamic translation techniques have been used to overcome the lack of flexibility inherent in a statically compiled simulation. The MIMIC simulator [5] simulates IBM System/370 instructions on the IBM RT PC. Groups of target basic blocks are translated to host instructions, with expansion factor of about 4 compared with natively compiled source code. On average MIMIC could simulate S/370 code at the rate of 200 instructions per second on a 2 MIPS RT PC.

Shade [6] and Embra [7] also used dynamic binary translation and translation caching, resulting in increased simulation speeds. Shade is able to simulate SPARC V8, SPARC V9, and MIPS I code on a SPARC V8 platform. On average Shade simulates V8 SPEC89 integer and floating-point binaries 6.2 and 2.3 times slower respectively than they run natively. The corresponding V9 binaries are simulated 12.2 and 4 times slower respectively. Embra on the other hand, which is part of the SimOS [8] simulation environment, can simulate MIPS R3000/R4000 binary code on a Silicon Graphics IRIX machine. In its fastest configuration Embra can simulate SPEC92 benchmarks at speeds ranging from 11.1 to 20 MIPS, corresponding to slowdowns of 8.7 to 3.5 when compared to



(a) (b) (c)

Fig. 9. Overall simulation performance for a selection of benchmark applications: chart (a) shows the simulation rate when JIT-translation is enabled; chart (b) shows the simulation rate when simulation is purely interpretive; chart (c) shows the distribution of time spent simulating and translating for each benchmark, and also the number of instructions executed by each benchmark.



(a) (b) (c)

Fig. 10. Dynamic simulation trace showing boot-up and operation of Linux: chart (a) shows the distribution of time spent performing binary translation versus simulating; chart (b) shows how the percentage of instructions simulated in interpretive and JIT-translated modes evolves during the simulation; chart (c) shows the instantaneous simulation rate.

native code. The simulation rate for the MAB benchmark is 5.6 MIPS representing a slowdown by a factor of 8.9. The test machine used for benchmarking was an SGI Challenge with four MIPS R4400, 150MHz processors.

More recently a number of research groups have developed retargetable instruction set simulators. The static compiled method in [9] applies static scheduling techniques to retargetable simulation, improving simulation performance at the expense of flexibility. Compiled simulators were generated from model descriptions of TI's TMS320C54x (cycle accurate) and the ARM7 (instruction accurate) processors. A FIR filter was used to benchmark both processor models running on an 800MHz Athlon PC. The results showed that for the TMS320C54X processor, static scheduling led to a speedup of almost a factor of 4 when compared to dynamically scheduled simulation. For the ARM7 processor there was an observed speedup by a factor of 7, from 5 (dynamic) to 35.5 (static) MIPS. The Ultra-fast Instruction Set Simulator in [10] improves the performance of static compiled simulation through the use of low-level binary translation techniques to take full advantage of the host architecture. Results showed that employing static compilation with this hybrid technique led to a 3.5-fold increase in simulation speed.

Just-In-Time Cache Compiled Simulation (JIT-CCS) [11] executes and then caches pre-compiled instruction-operation functions for each instruction fetched. The speed of simulation of the JIT-CCS simulator with a reasonably large simulation cache is within 5% of a compiled simulator, and at least 4 times faster than an equivalent interpretive simulator. The simulator was benchmarked by simulating adpcm running on ARM7 and STM ST200 functional models. Results showed that adpcm simulated at up to 8 native MIPS for the ARM7 and ST200, running on a 1200MHz Athlon host.

The Instruction Set Compiled Simulation (IS-CS) simulator [12] 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. Performance is further increased by a technique called instruction abstraction which produces optimized decoded instructions. A simulation rate of up to 12.2 MIPS is quoted for adpcm on an ARM7 functional model running on a 1GHz Pentium III host.

The SimICS [13] full system simulator translates the target machine-code instructions in to an intermediate format before interpretation. During simulation the intermediate format instructions are processed by the interpreter which calls

the corresponding service routines. A number of SPECint95 benchmarks running under SunOS 5.x were simulated using SimICS and a Sun Ultra Enterprise host. The results showed a performance decrease by a factor of 26 to 75 compared with native execution of the benchmarks.

SyntSim is a synthesis system for functional simulators which uses binary translation to achieve high performance [14]. SyntSim performs static, or off-line, binary translation based on an optional application profile and the application binary. The authors report an average slowdown of 6.6 compared with native execution, on SPECcpu2000 benchmarks. Our simulator goes beyond this in several important respects. Firstly we perform a truely dynamic binary translation, allowing the system to cope with self-modifing code and the swapping behavior of multi-tasking operating systems such as Linux. Secondly, our simulation retains a precise view of the target processor state at all points of observation, allowing the simulator to be used as a high speed debugger target or functional verification engine. Thirdly, we model the TLBs and realistic I/O devices, to support real operating systems and multi-threaded workloads.

X. Conclusions

This paper presents a full-system simulator which is fast enough to simulate many embedded applications in real time, whilst also offering the precise observability required from the golden reference model of an embedded system. By linking together these two capabilities in the same simulator, we are able to support the seemingly disparate requirements of processor verification, embedded software development, system performance evaluation, and design-space exploration in a single tool.

References

[1] C. Mills, S. C. Ahalt, and J. Fowler, "Compiled instruction set simulation," *Software, Practice and Experience*, vol. 21, no. 8, pp. 877–889, 1991.

[2] *ARCompact™ Instruction Set Architecture*, ARC International, 2025 Gateway Place, Suite 140, San Jose, CA 95110, USA. [Online]. Available: http://www.arc.com/

[3] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," *SIGARCH Comput. Archit. News*, vol. 25, no. 3, pp. 13–25, 1997.

[4] D. Ung and C. Cifuentes, "Dynamic binary translation using run-time feedbacks," *Sci. Comput. Program.*, vol. 60, no. 2, pp. 189–204, 2006.

[5] C. May, "Mimic: a fast system/370 simulator," in *SIGPLAN '87: Papers of the Symposium on Interpreters and interpretive techniques*. New York, NY, USA: ACM Press, 1987, pp. 1–13.

[6] B. Cmelik and D. Keppel, "Shade: a fast instruction-set simulator for execution profiling," in *SIGMETRICS '94: Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM Press, 1994, pp. 128–137.

[7] E. Witchel and M. Rosenblum, "Embra: fast and flexible machine simulation," in *SIGMETRICS '96: Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM Press, 1996, pp. 68–79.

[8] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta, "Complete computer system simulation: The simos approach," *IEEE Parallel Distrib. Technol.*, vol. 3, no. 4, pp. 34–43, 1995.

[9] G. Braun, A. Hoffmann, A. Nohl, and H. Meyr, "Using static scheduling techniques for the retargeting of high speed, compiled simulators for embedded processors from an abstract machine description," in *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*. New York, NY, USA: ACM Press, 2001, pp. 57–62.

[10] J. Zhu and D. D. Gajski, "A retargetable, ultra-fast instruction set simulator," in *DATE '99: Proceedings of the conference on Design, automation and test in Europe*. New York, NY, USA: ACM Press, 1999, p. 62.

[11] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann, "A universal technique for fast and flexible instruction-set architecture simulation," in *DAC '02: Proceedings of the 39th conference on Design automation*. New York, NY, USA: ACM Press, 2002, pp. 22–27.

[12] M. Reshadi, P. Mishra, and N. Dutt, "Instruction set compiled simulation: a technique for fast and flexible instruction set simulation," in *DAC '03: Proceedings of the 40th conference on Design automation*. New York, NY, USA: ACM Press, 2003, pp. 758–763.

[13] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenstrm, and B. Werner, "Simics/sun4m: A virtual workstation," 1998. [Online]. Available: citeseer.comp.nus.edu.sg/magnusson98simicssunm.html

[14] M. Burtscher and I. Ganusov, "Automatic synthesis of high-speed processor simulators," in *Proceedings of the 37th Annual International Symposium on Microarchitecture*, Dec. 2004, pp. 55–66.