# FaCSim: A Fast and Cycle-Accurate Architecture Simulator for Embedded Systems *

Jaejin Lee†, Junghyun Kim†, Choonki Jang†, Seungkyun Kim†,
Bernhard Egger‡, Kwangsub Kim+, and SangYong Han†
http://aces.snu.ac.kr

†School of Computer Science and Engineering
Seoul National University
Seoul 151-744, Korea
jlee@cse.snu.ac.kr
{junghyun, choonki, seungkyun}@aces.snu.ac.kr
syhan@pandora.snu.ac.kr

‡Samsung Institute of Technology
Giheung-gu, Yongin-si
Gyeonggi-do 446-712, Korea
bernhard@aces.snu.ac.kr

+Digital Media Research Lab.
LG Electornics
Woomyun-dong, Seocho-gu
Seoul 137-724, Korea
whatsub@lge.com

## Abstract

There have been strong demands for a fast and cycle-accurate virtual platforms in the embedded systems area where developers can do meaningful software development including performance debugging in the context of the entire platform. In this paper, we describe the design and implementation of a fast and cycle-accurate architecture simulator called FaCSim as a first step towards such a virtual platform. FacSim accurately models the ARM9E-S processor core and ARM926EJ-S processor's memory subsystem. It accurately simulates exceptions and interrupts to enable whole-system simulation including the OS. Since it is implemented in a modular manner in C++, it can be easily extended with other system components by subclassing or adding new classes. FaCSim is based on an interpretive simulation technique to provide flexibility, yet achieving high speed. It enables fast cycle-accurate architecture simulation by means of three mechanisms. First, it computes elapsed cycles in each pipeline stage as a chunk and incrementally adds it up to advance the core clock instead of performing cycle-by-cycle simulation. Second, it uses a basic-block cache that caches decoded instructions at the basic-block level. Finally, it is parallelized to exploit multicore systems that are available everywhere these days. Using 21 applications from the EEMBC benchmark suite, FaCSim's accuracy is validated against the ARM926EJ-S development board from ARM, and is accurate in a $\pm 7\%$ error margin. Due to basic-block level caching and parallelization, FaCSim is, on average, more than three times faster than ARMulator and more than six times faster than SimpleScalar.

***Categories and Subject Descriptors*** C.4 [*Performance of Systems*]: Modeling Techniques

***General Terms*** Algorithms, Design, Measurement, Performance, Experimentation

***Keywords*** Architecture Simulator, Simulator Parallelization, Cycle-Accurate Simulation, Full-System Simulation, Virtual Prototyping

* This work was partly supported by the IT R&D program of MIC/IITA [2006-S-040-01, Development of Flash Memory-based Embedded Multimedia Software] and by the Ministry of Education, Science and Technology under the BK21 Project. ICT at Seoul National University provided research facilities for this study.



*LCTES'08,* June 12–13, 2008, Tucson, Arizona, USA.


## 1. Introduction

One difficulty in developing embedded software is that the target hardware platform is often still under development in parallel due to the pressure of speeding time to market. Moreover, the trends in digital convergence require many different functionalities combined in a single consumer electronics or mobile device. Consequently, demands for computing power in embedded systems have been continuously increased. Therefore, performance debugging, in addition to traditional functional debugging, plays a key role in embedded software development. As a result, there are strong demands for a fast and cycle-accurate virtual platform, where developers can do meaningful software development in the context of the entire platform.

In this paper, we describe the design and implementation of a fast and cycle-accurate architecture simulator, FaCSim, as a first step towards a fast and cycle-accurate virtual platform for simulating the whole system including operating systems and peripherals. There are some proposals for full system simulators [20, 27, 30]. These simulators focus on functional accuracy with high performance rather than cycle-level accuracy.

FaCSim models the ARM9E-S processor's pipeline and ARM926EJ-S processor's memory subsystem including the instruction and data caches, instruction and data scratchpad memory, instruction and data micro-TLBs, unified TLB, MMU, write buffer, prefetch buffer, bus, and main memory. Since one of the most important aspects of full system simulation is accurately modeling the virtual memory system, it precisely models the memory management unit in the ARM926EJ-S core and accurately simulates exceptions and interrupts. Since FaCSim is implemented in a modular manner in C++ from scratch, it is easy to extend FaCSim with other architecture components by subclassing them or adding new classes with the same interface.

FaCSim uses an interpretive simulation technique to provide flexibility and high accuracy, yet achieving high speed simulation. We achieve fast and cycle-accurate simulation through three mechanisms. First, FaCSim computes the number of elapsed cycles in

each pipeline stage as a chunk and incrementally adds it up to advance the core clock instead of performing cycle-by-cycle simulation. Second, it caches decoded instructions at the basic block level. Since each decoded instruction is a C++ object in FacSim, creating and destroying the instruction objects is a serious performance bottleneck. The block-level caching technique alleviates this bottleneck. This is a well-known technique in the compiled simulation area, but previously not applied to interpretive simulators. Finally, it is parallelized to exploit multicore systems that are available everywhere these days. FaCSim is divided into a functional front-end and a cycle-accurate back-end, and they run in parallel on two tightly-coupled cores. It uses a non-deterministic producer and consumer algorithm that completely eliminates synchronization between the two cores.

We validate FaCSim's accuracy by comparing it to ARMulator [17] and the ARM926EJ-S development board [16] with 21 applications from the EEMBC benchmark suite. FaCSim's core pipeline is almost 100% accurate compared to ARMulator. The accuracy of the entire simulator is validated against the development board, and is accurate in a $\pm 7\%$ error margin. Due to basic-block level caching and parallelization, FaCSim is, on average, more than three times faster than ARMulator and six times faster than SimpleScalar [34].

The rest of this paper is organized as follows: Section 2 discusses related work. Section 3 describes the design and implementation of FaCSim. We describe the evaluation environment in Section 4, and Section 5 presents the evaluation results. The paper concludes in Section 6.

## 2. Related Work

In the last two decades, there have been many studies done on instruction set simulators (ISS) to improve their performance. Many ISSs are based on interpretive simulation techniques, where the simulator is looping in the fetch-decode-execute cycle. For such simulators, the decoding overhead is a major bottleneck to performance. To remove the decoding overhead of interpretive simulators, Mills et al. [22] proposed the static compiled simulation technique, where the input application binary is translated into C code that is then compiled with the simulator source code. Even though there are many proposals of static compiled simulation [2, 8, 11, 18, 22, 24, 29, 40] to improve simulation performance, it has two major drawbacks. One is that the compilation time may take longer than the actual simulation time for large applications. The other is that it cannot handle self-modifying code. To overcome these drawbacks, people proposed dynamic compiled simulation [1, 4, 26, 27, 28, 31, 32, 38], which is based on just-in-time compilation techniques that exploit code caches. However, the compiled simulation techniques either translate the target instructions into the native instruction set of the host system, or assume a virtual machine whose instruction set architecture (ISA) is different from the target ISA. Consequently, it cannot exactly model the target architecture, such as pipeline interlocks, resulting in loosing accuracy of application execution time. Most of the simulators that claim cycle accuracy, such as SimpleScalar [34], ARMulator [17], and SESC [33], are based on interpretive simulation.

Some proposals [6, 12, 20, 25, 28, 30, 37, 38] target multiprocessor host system for high speed simulation or profiling. Lauterbach et al. [12] proposed executing multiple independent trace samples from an application in parallel. Qin et al. [28] proposed a parallelized compiled simulation technique. One processor generates the dynamic linking library for a part of the application and another processor runs it during its interpretive simulation. SuperPin [37] is a parallel version of Pin[19]. One processor generates

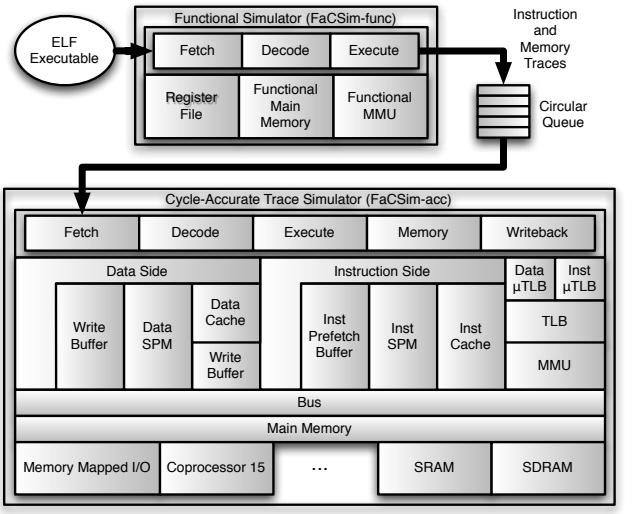


**Figure 1.** The structure of FaCSim.

instrumentation slices of an application, and other processors run the slices for instrumenting the entire application. Other proposals [6, 20, 25, 30, 38] exploit the multiprocessor host system by running multiple single processor/core simulators on different host processors/cores to perform multiprocessor/multicore architecture simulation.

FaCSim is different from the above parallelization approaches in that the simulator for a single processor system is parallelized into two parts that run on two different cores which are tightly coupled by a shared cache. FaCSim exploits a well-known functional model/timing model decoupling [3, 7, 21, 23, 31], and is divided into a functional front-end and a cycle-accurate back-end that run in parallel. FaCSim's parallelization approach is similar to FAST[3], where the timing model is implemented in an FPGA to boost performance over a software implementation. The FPGA implementation enables FAST to exploit the parallelism between modules within the timing model and between the timing model and the functional model. However, the FPGA implementation is lacking observability into the microarchitecture modeled and flexibility of modification compared to the software implementation. The timing model in FaCSim is implemented purely in software, and FaCSim still achieves high performance. FaCSim uses a non-deterministic producer/consumer algorithm that completely eliminates synchronization between the two ends and boosts performance.

There are some whole system simulators, such as SimOS [30], QEMU [27], and Simics [20]. They can run unmodified commercial operating system images. These simulators focus on the functional accuracy with high performance rather than the cycle-level accuracy.

## 3. FaCSim Design and Implementation

In this section, we describe the design and implementation of FaCSim in detail. We also describe the techniques used in FaCSim to achieve high accuracy and simulation speed.

### 3.1 The Overall Structure

FaCSim is intended for fast full system simulation with cycle-level accuracy. FaCSim models the ARM9E-S core [14] and a memory hierarchy including the MMU implemented in the Versatile Application Baseboard (AB) for ARM926EJ-S from ARM [13, 15, 16].

It is able to run unmodified applications and the Linux operating system that target the Versatile AB. Figure 1 shows the overall structure of FaCSim. FaCSim basically consists of two parts: a functional simulator (FaCSim-func) and a cycle-accurate trace simulator (FaCSim-acc). The functional simulator fetches an instruction word, decodes it, and executes it. It generates execution traces that are buffered in a circular queue located in-between FaCSim-func and FaCSim-acc. FaCSim-acc simulator fetches the traces and performs cycle-accurate simulation. FaCSim is implemented in C++ from scratch, and all machine components and instructions are represented in classes and objects in FaCSim.

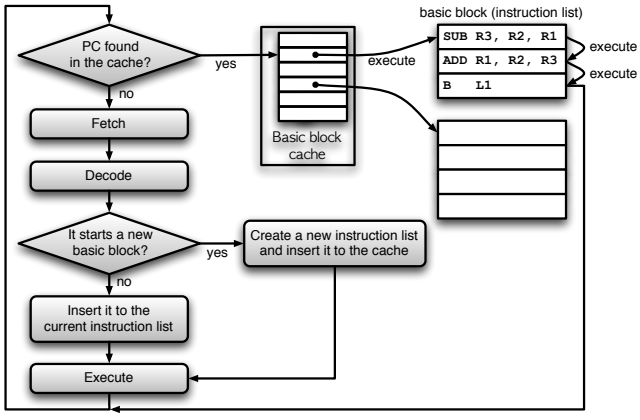


**Figure 2.** Caching instruction objects at the basic-block level.

### 3.2 The Functional Simulator

FaCSim-func models system components that hold the machine state or that are not functionally transparent to the user, such as a register file, main memory, and the MMU.

**Instruction abstraction.** In FaCSim-func, instructions with similar formats are classified into the same class in C++. This enables separate specialized functions for each class of instructions. In turn, it increases the chance of optimization by the compiler compared to defining a generic class for all instruction formats. We follow the instruction classification by ARM [15]: branch, data-processing, multiply, miscellaneous arithmetic, status register access, load and store, load and store multiple, semaphore, exception-generating, and coprocessor instructions. These classes are the subclasses of the top abstract base class `Instruction`, and they implement a virtual member function `execute` that changes the state of the machine (e.g., the registers and main memory) appropriately according to the specific task of each subclass.

FaCSim-func performs the fetch-decode-execute cycle on each instruction word in the application. After fetching an instruction word and decoding it, FaCSim-func creates an instruction object. By invoking the `execute` method of the object, FaCSim-func executes the instruction and generates a corresponding instruction trace object. After generating the trace object, it destroys the instruction object.

**Caching instruction objects.** Since creating and destroying an instruction object incurs a lot of overhead whenever an instruction is executed, FaCSim-func provides two different object caching mechanisms: one is at the individual instruction level and the other is at the basic-block level.

The instruction-level object caching provides a direct-mapped cache for decoded instruction objects. The cache is indexed by the virtual address of the instruction word to be fetched. On a hit, the cache entry contains the pointer to the instruction object. FaCSim-func skips the fetch and decode steps and directly invokes the `execute` method of the instruction object. Otherwise, it fetches and decodes the instruction word, and inserts the newly created instruction object into the cache. Then it executes the instruction by invoking its `execute` method.

The basic-block level object caching further reduces the overhead of creating and destroying instruction objects in the decode step. It is based on the observation that if the first instruction in the basic block is executed, then the rest of the instructions in the same block are also executed.

A basic block is implemented with a list for caching, and it is identified by the virtual address of its first instruction. Figure 2 illustrates the mechanism of caching objects at the basic-block level. After decoding an instruction word, FaCSim-func checks whether it belongs to the same block as that of the previous instruction. If so, it adds the current instruction object to the current instruction list. Otherwise, it inserts the current instruction list in the direct-mapped cache for the basic blocks. An entry in the cache points to an instruction list. When FaCSim-func is given the virtual address of an instruction word to be fetched, it checks the cache with the address. On a hit, the instruction is the first instruction of a basic block. FaCSim-func takes the entire block and executes all instructions in the list one by one. On a miss, the instruction is the first instruction of a new basic block. After creating a new list and inserting the instruction object in it, FaCSim-func goes through the normal fetch-decode-execute cycle for the instruction.

**Modeling exceptions.** FaCSim-func supports exceptions generated by internal and external sources, such as attempts to execute an undefined instruction or external interrupts. All seven types of exceptions in the ARM architecture (i.e., reset, undefined instructions, software interrupt, prefetch abort, data abort, interrupt, and fast interrupt) are accurately modeled by FaCSim. When an exception occurs, FaCSim-func switches the processor mode appropriately according to the type of the exception and starts generating instruction traces for the exception handler if it is ever provided by the software being simulated. It also supports privileged instructions that can only be executed in a specific privileged mode. FaCSim-func completely models all the processor modes and their register banks supported by the ARM9E-S core.

When an exception is generated while executing an instruction that is a member of a cached basic block, it invalidates the entire block from the object cache to precisely model the exception.

**Modeling MMU address translation.** FaCSim-func models the functions of the virtual memory system architecture as described in [15]. The functional MMU is used for every instruction fetch and every data access to translate the virtual addresses into physical addresses. The translation also checks access permissions and determines whether the access is bufferable and cacheable.

There are four types of address translation and permission check in the ARM architecture depending on whether the address has been mapped as a section or a page. Sections are comprised of 1MB blocks of memory, and the section descriptor (i.e., the page table entry) is obtained by accessing the first-level page table. A page-mapped access can be a large (64KB), small (4KB), or tiny (1KB) page access. The page table descriptor for page-mapped accesses is obtained by accessing the second-level page table, resulting in two memory accesses.

Whenever there is a memory access by an instruction or due to an instruction fetch, FaCSim-func translates the virtual address into

the physical address using the functional MMU, and obtains the section or page table descriptor depending on the address mapping. If there is no access violation (i.e., a permission violation or a page fault), it generates a table walk trace, which contains the physical addresses accessed by the page table walk algorithm, in addition to a memory trace for the requested virtual address. If there is an access violation, FaCSim-func generates an exception. The exceptions generated by the functional MMU are prefetch aborts and data aborts.

FaCSim-func does not model TLBs because their function is transparent to the software being simulated, and it affects only execution time. Their effects on execution time is modeled in the cycle-accurate back-end, FaCSim-acc. Similarly, the instructions for locking TLB entries affect only the cycle-accuracy of the simulation. Even though these instructions do not affect functional correctness, they are passed to FaCSim-acc as instruction traces to allow cycle-accurate simulation.

**Self-modifying code.** Since the original instruction words are not fetched again once a block of objects is cached, self-modifying code is a problem. If the code has changed, FaCSim-func would execute the stale version of the instructions. Therefore, FaCSim-func keeps track of pages that have instructions in the basic-block cache. When any of these pages is overwritten, FaCSim-func flushes the entire basic-block cache.

### 3.3 Execution Traces

FaCSim-func generates a sequence of table walk, memory, and instruction traces as a result of the functional simulation. These execution traces are an intermediate representation of the execution of the software running on the simulator. They are implemented in C++ objects. A table walk trace is contained in a memory trace. In turn, multiple memory traces are contained in an instruction trace. The instruction traces are saved in a circular queue and passed to FaCSim-acc.

**Table walk traces.** A table walk trace consists of two physical addresses of two page table entries. Only one of them is valid if the requested memory access is a section reference. Otherwise, FaCSim-acc uses both of them to compute memory latencies taken in accessing both the first-level and second-level page tables. The table walk trace class includes a member function `commit` that is invoked by FaCSim-acc. It takes as an argument the value of the core clock at the time when the table walk occurs, and returns the value plus the latency caused by the table walk:

```
virtual SValue64 commit( SValue64 clock_start );
```

**Memory traces.** The topmost base class of memory traces includes five member variables: virtual address, physical address, bufferable bit, cacheable bit, and a pointer to a table walk trace object. The values of the first four members are obtained using the functional MMU in FaCSim-func for the requested memory access. The bufferable bit tells whether the memory access can be buffered in the write buffer, and the cacheable bit tells whether it is cacheable. The base class also includes a virtual member function `commit` that takes as an argument the value of core clock at the time when the memory access occurs, and returns the value plus the latency taken by the memory access:

```
virtual SValue64 commit( SValue64 clock_start );
```

It is called by FaCSim-acc for cycle-accurate simulation, and each subclass of memory traces implements the method appropriately. A specific subclass is defined for each type of memory accesses: instruction fetch, word read/write, halfword read/write, byte read/write, four-word burst read/write, eight-word burst read/write, atomic word swap, and atomic byte swap.

**Instruction traces.** The topmost base class of instruction traces includes two member variables: the execution latency and a pointer to an instruction fetch memory trace object. The execution latency of an instruction is defined by the number of core clock cycles spent by the instruction in the execute stage of the ARM9E-S pipeline. The latency of each instruction is obtained from [14]. The class also includes three virtual member functions that are called by FaCSim-acc in each stage of the ARM9E-S pipeline for cycle-accurate simulation except for decode and writeback stages:

```
virtual SValue64 fetch( SValue64 clock_fetch );
virtual SValue64 execute( SValue64 clock_exe );
virtual SValue64 memory( SValue64 clock_mem );
```

The method for each stage takes as an argument the value of the core clock at the time when the instruction trace enters into that stage, and returns the value of the core clock plus its latency in that stage. Since the latency of each instruction in the decode and writeback stages is a fixed constant, i.e., one cycle, there are no methods defined for those stages. The virtual functions are implemented by the subclasses of the instruction traces. The instruction traces are divided into subclasses based on their type and the way of computing their execution latency.

- **Compute instruction traces.** Multiply, miscellaneous arithmetic, status register access, and data-processing instructions. An object in this subclass contains no other memory trace object than an instruction fetch trace.
- **Branch instruction traces.** Branch and data-processing instructions that modify the program counter (PC). If FaCSim-func meets a branch or data-processing instruction that modifies the PC (i.e., a branch instruction), it generates two instruction fetch instruction traces that follow the branch instruction trace to correctly simulate the core pipeline. The instruction trace for the branch target is generated after the two instruction fetch instruction traces.
- **Memory instruction traces.** Load/store and load/store multiple instructions that do not modify the PC. An object in this subclass contains at least one memory trace object for data accesses.
- **Memory and Branch instruction traces.** Load/store and load/store multiple instructions that modify the PC. The behavior of this subclass is similar to the branch instruction trace, but an object in this subclass contains at least one memory trace object for data access.
- **Semaphore instruction traces.** Semaphore instructions, such as swap word and swap byte instructions.
- **Exception generating instruction traces.** Instruction traces generated by software interrupt instructions.
- **Coprocessor instruction traces.** Instruction traces generated by the instructions that access coprocessor 15 control registers. If their `execute` method is invoked by FaCSim-acc, it changes the machine configuration as a result.

FaCSim-func generates the extra instruction traces below to correctly reflect the effect of fetched but not executed instructions, pipeline interlocks, and fetched but invalidated instructions in the pipeline due to a branch.

- **Unexecuted instruction traces.** Instruction traces generated by instructions that are fetched but not executed. Their execution latency is just one cycle.
- **Instruction fetch instruction traces.** Instruction traces generated by the instructions that modify the PC. Since these instructions do not go through the execute, memory, and writeback stage of the ARM9E-S pipeline, their execution latency is zero.

- **Bubble instruction traces.** Instruction traces generated to resolve pipeline interlocks. Their execution latency is just one cycle. If FaCSim-func identifies interlocks between the preceding instructions and the current instruction being executed, it inserts an appropriate number of bubble instruction traces in front of the current instruction.

Since there is a significant overhead involved to create these execution trace objects whenever an instruction is executed, FaCSim-func creates enough execution trace objects for each subclass before the simulation begins. Instead of destroying them after being used by FaCSim-acc, FaCSim-func reuses them.

**Pipeline interlocks.** FaCSim-func keeps track of dependences between instructions. It detects a dependence between the current ($n^{th}$) instruction and the previous (n-$1^{th}$) instruction. If the dependence incurs a $k$-cycle pipeline interlock, FaCSim-func generates $k$ bubble instruction traces in between the two dependent instructions to correctly model pipeline interlocks at the FaCSim-acc side. FaCSim-func also detects a dependence between the $n^{th}$ and the n-$2^{th}$ instructions, and inserts an appropriate number of bubble instruction traces depending on the number of bubble instructions generated by the dependence between the $n^{th}$ and the n-$1^{th}$. Pipeline interlocks occur mainly due to lacking forwarding paths and operand buses in the ARM9E-S core. The cases of pipeline interlocks in ARM9E-S are well explained in the documentation [14].

FaCSim-func contains two tables that record the number of pipeline interlocks for different forms of dependences. One table is indexed by a pair that consists of the type of the n-$1^{th}$ instruction and that of the $n^{th}$. The other is indexed by a pair that consists of the types of the n-$2^{th}$ and the $n^{th}$.

### 3.4 The Cycle-Accurate Trace Simulator

FaCSim-acc is the cycle-accurate trace simulator. It computes the elapsed cycles in each pipeline stage as a chunk and incrementally adds them up to advance the core clock instead of performing cycle-by-cycle simulation. It consists of a five-stage pipeline, instruction and data caches, instruction and data scratchpad memory, instruction and data micro-TLBs, a unified TLB, an MMU, a write buffer, an instruction prefetch buffer, a bus, and main memory. It models with C++ classes those components that can be found in ARM9EJ-S [13] and the Versatile AB for ARM926EJ-S [16].

**Modeling the pipeline.** FaCSim-acc fetches an instruction trace from the circular queue, and makes it go through the five-stage pipeline. Each stage in the pipeline invokes its corresponding method from the instruction trace to obtain the delay in that stage. Since the latency of an instruction in the decode and writeback stage is a fixed constant (i.e., one cycle), there is no method invocation in these stages. The delay obtained in each stage is reflected on the core clock. FaCSim-acc computes the core clock at the beginning of each stage, say $clock\_in$, (i.e., when the instruction trace just enters into the stage) and at the end of each stage, say $clock\_out$, (i.e., when the instruction just leaves the stage) starting from the writeback stage to the fetch stage in turn. The core clock computed at the end of the writeback stage is used to advance the core clock cycles.

Basically, $clock\_out$ of each stage is computed by the assignments below. $Latency\_X$ stands for the delay caused by the instruction in stage $X$. The latencies $latency\_M$, $latency\_E$, and $latency\_F$ vary depending on the instruction type and the state of the memory hierarchy.

```
            ...
0x8400    MOV  r1, #2
0x8404    MUL  r0, r1, r2
0x8408    LDR  r2, [r0, #4]
0x840C    SUBS r4, r0, r3
0x8410    BEQ  L1
0x8414    ADD  r0, r1, r2
0x8418    ADD  r0, r0, #1
            ...
0x8700    MOV  r0, #0
```

**Figure 3.** A sequence of instructions.

$$
\begin{aligned}
clock\_out\_W &:= clock\_in\_W + 1 \\
clock\_out\_M &:= clock\_in\_M + latency\_M \\
clock\_out\_E &:= clock\_in\_E + latency\_E \\
clock\_out\_D &:= clock\_in\_D + 1 \\
clock\_out\_F &:= clock\_in\_F + latency\_F
\end{aligned}
$$

After $clock\_out$ at each stage is obtained, $clock\_in$ for the next stage is computed by the assignments below in the order they appear. $Clock\_out\_X$ stands for $clock\_out$ of stage $X$.

$$clock\_in\_W := clock\_out\_M$$

$$clock\_in\_M := MAX(clock\_out\_E, clock\_in\_W)$$

$$clock\_in\_E := MAX(clock\_out\_D, clock\_in\_M)$$

$$clock\_in\_D := MAX(clock\_out\_F, clock\_in\_E)$$

$$clock\_in\_F := clock\_in\_D$$

Executing a branch instruction or branching by modifying the PC causes the pipeline to be flushed. That is, the two instructions that follow the branch instruction will be fetched and then flushed from the pipeline. Then, $clock\_in\_F$ for the target instruction is the core clock at the time when the PC-modifying instruction fetches its target instruction in the execute stage.

**Modeling the memory hierarchy.** The components in the memory hierarchy of FaCSim-acc do not update their contents. They just return their latencies to FaCSim-acc when they are accessed by table walk traces or memory traces. There are three different paths from the core to the bus in the memory hierarchy: data side, instruction side, and table walk. Each component in the paths is a C++ class. The components on the same path share a common interface that is invoked by either a table walk trace or a memory trace. Consequently, a new component in the memory hierarchy can be easily added to one of the paths if the new component implements the same common interface for the path.

**Modeling the bus.** FaCSim-acc models a multi-layer AHB system [13, 16], with two separate AHB masters for the instruction side and the data side. If the two AHB interfaces require access to the same slave at the same time, then an arbitration process is needed. Under the current implementation of FaCSim-acc, the data side has a higher priority than the instruction side. The bus class implements the two interfaces that are invoked by the instruction-side components and the data-side components.

The bus clock is synchronized to the core clock whenever there is a bus request (i.e., a call to a method in the bus class). A bus request has the value of the core clock, say $C_{core}$, at the time when the request has been made. If $r$ times the value of the bus clock is less than $C_{core}$, the bus clock advances to $\lceil C_{core}/r \rceil$ assuming $r$ is

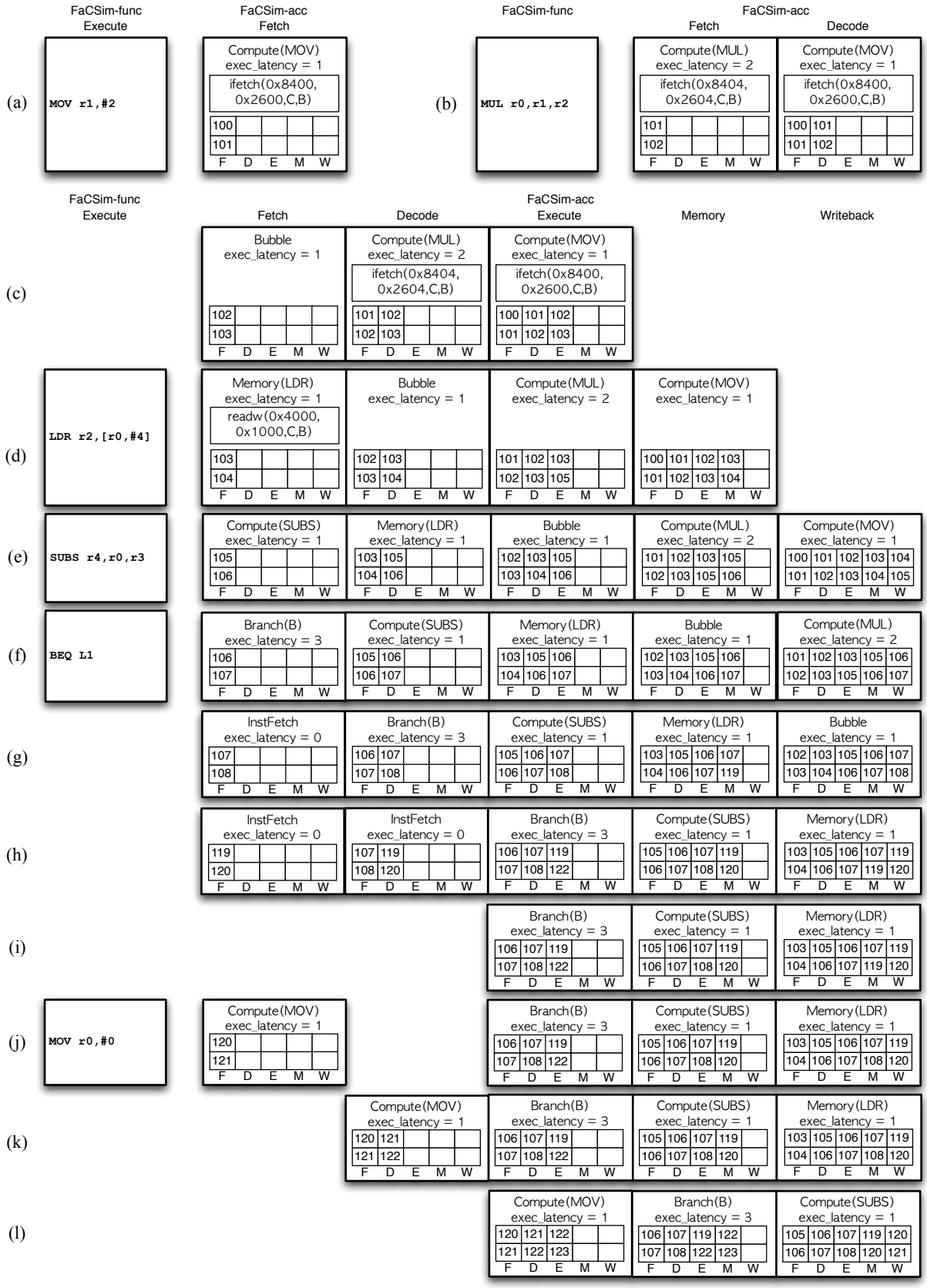


**Figure 4.** An example of FaCSim simulation.

the ratio $\frac{core\ clock}{bus\ clock}$. Otherwise there is no change in the bus clock. Then, the bus invokes the target slave component. The latency returned by the target slave component is added to the bus clock, and the bus clock advances. The value of the bus clock is returned to the requester.

### 3.5 An Example of FaCSim Simulation

Consider the instruction sequence as shown in Figure 3. The address presented in front of each instruction is its virtual address. In Figure 4(a), FaCSim-func executes `MOV r1,#2`, generates a *compute* instruction trace Compute(MOV) whose latency in the execute stage is one. It also contains an instruction fetch memory trace ifetch(0x8400, 0x2600, C, B). 0x8400 is the virtual address of the instruction and 0x2600 is its physical address. The capital letter C stands for the fact that the instruction is cacheable and B for the fact that it is bufferable. For simplicity, we omit the table walk trace in the memory trace. The generated instruction trace is stored in the circular buffer. FaCSim-acc fetches the instruction trace from the buffer and inserts it into the fetch stage of the pipeline. Assume that $clock\_in$ of the fetch stage is 100. FaCSim-acc makes the instruction fetch memory trace ifetch(0x8400, 0x2600, C, B) go through the MMU and the memory hierarchy. Assume that the table walk of the fetch trace produces a hit in the instruction $\mu$TLB, and the fetch is also an instruction cache hit with a total of one cycle latency. Thus, $clock\_out\_F$ is equal to 101.

FaCSim-func executes the next instruction `MUL r0,r1,r2` in Figure 4(b). It generates a compute instruction trace Compute(MUL) with the execution latency 2 and the instruction fetch memory trace ifetch(0x8404, 0x2604, C, B). FaCSim-acc fetches Compute(MUL) from the circular queue and inserts it into the cycle-accurate pipeline. Since the previous instruction trace moves on to the decode stage at clock 101, $clock\_in\_F$ is 101 for Compute(MUL). Assume that fetching Compute(MUL) is a cache hit. Thus, $clock\_out\_F$ becomes 102. In addition, $clock\_out\_D$ is equal to 102 because the latency in the decode stage is 1. Since there is a dependence between `MUL r0,r1,r2` and the next instruction `LDR r2,[r0,#4]`, and the result of `MUL r0,r1,r2` is only available at the end of the memory stage, there will be a one-cycle pipeline interlock. FaCSim-func generates a bubble instruction trace for this pipeline interlock in Figure 4(c). FaCSim-acc fetches this bubble instruction trace from the circular queue, and inserts it into the cycle-accurate pipeline. Now, Compute(MOV) is in the execution stage, and $clock\_in\_E = 102$ and $clock\_out\_E = 103$ because the execution latency of Compute(MOV) is equal to 1.

FaCSim-func executes the next instruction `LDR r2,[r0,#4]` and generates a memory instruction trace Memory(LDR) in Figure 4(d). This trace contains an instruction fetch memory trace ifetch(0x8408, 0x2608, C, B) and a memory trace readw(0x4000, 0x1000, C, B). FaCSim-acc inserts Memory(LDR) into the pipeline. Currently, $clock\_in\_E = 103$ and $clock\_out\_E = 105$ because the execution latency of Compute(MUL) is equal to 2. Then, FaCSim-func executes the next instruction `SUBS r4,r0,r3` and generates a compute instruction trace Compute(SUBS) in Figure 4(e). The instruction trace of the first instruction `MOV r1,#2` is graduated from the cycle-accurate pipeline when the core clock is 105.

FaCSim-func executes the branch instruction `BEQ L1` at 0x8410 and generates a branch instruction trace Branch(B) in Figure 4(f). Assume that `BEQ L1` is a taken branch. Since it is taken, FaCSim-func generates two instruction-fetch instruction traces for the two instructions (i.e., `ADD r0,r1,r2` at 0x8414 and `ADD r0,r0,#1` at 0x8418) that follow `BEQ L1` in the code (Figure 4(g) and Figure 4(h)). In Figure 4(g), Memory(LDR) reaches the memory stage when the core clock is equal to 107. FaCSim-acc makes the memory trace readw(0x4000,0x1000,C,B) in Memory(LDR) go through the memory hierarchy. Assume that it is a data cache miss and takes 12 cycles. The load instruction finishes its memory stage when the core clock is equal to 119. Then, the invocation of `execute()` of the instruction trace Branch(B) by FaCSim-acc makes the two instruction fetch traces to be flushed from the pipeline in Figure 4(i).

FaCSim-func executes the target instruction `MOV r0,#0` of the branch instruction and generates the trace Compute(MOV) in Figure 4(j). FaCSim-acc makes Compute(MOV) go through the fetch and decode stage. Note that $clock\_in\_F$ is equal to 120 because the branch instruction in the execute stage fetches its target when the core clock is equal to 120 (Figure 4(j) and Figure 4(k)). Finally, Compute(MOV) reaches its execute stage in Figure 4(l).

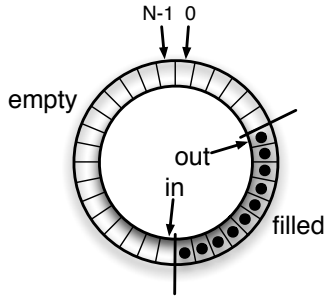


**Figure 5.** The circular queue.

### 3.6 Parallelization

In our design of FaCSim, FaCSim-func (the producer) and FaCSim-acc (the consumer) are well decoupled through the shared circular queue. Each entry of the queue is a pointer to an instruction trace object. This is a typical producer-consumer model. Since there are multicore host systems available everywhere these days, we parallelize FaCSim to exploit these multicore systems to improve its performance. In the parallelized version, FaCSim-func and FaCSim-acc are structured as two different threads that run on two different cores. The circular queue is the only shared data structure between the two threads.

In order to increase the degree of concurrency and to remove the synchronization overhead on the circular queue, we use a non-deterministic algorithm [10, 36, 39] that eliminates synchronization overhead significantly between the producer and consumer.

In FaCSim, the shared circular queue is implemented with an N-element array (Figure 5), and an entry in the queue points to an instruction trace generated by FaCSim-func. The range of available pointers in the queue is specified with two integer variables `in` and `out`. The variable `in` points to the location in the buffer where the next trace goes in (thus, the entry pointed by `in` is always empty even though the queue is full), and `out` points to the location where the next trace is fetched by FaCSim-acc. The variable `out` is modified only by FaCSim-acc and the variable `in` is modified only by FaCSim-func. They are incremented in a circular manner (i.e., modulo N).

To explain our algorithm in Figure 6, we denote the range of the circular queue $Q$ with two integers $x$ and $y$ in the following way:

$$Q[x,y) = \begin{cases} \{x \bmod N, (x+1) \bmod N, ..., (y-1) \bmod N\} & \text{if } x < y \\ \emptyset & \text{if } x = y \\ \{0,1,...,N-1\} - Q[y,x) & \text{if } x > y \end{cases}$$

```
while (...) {
   ...
   while (true) {
      Cin = in;
      Cout = out;
      if (Cin ≠ Cout) break;
   }
   Copy the instruction trace pointers stored in
         the circular queue in the range Q[Cout, Cin)
         to the local buffer;
   Cout = (Cout + # of pointers copied) mod N;
   out = Cout;
   ...
   Peform cycle-accurate simulation with the trace
         pointers stored in the local buffer
   ...
}
```

(a)

```
while (...) {
   ...
   while (true) {
      Pin = in;
      Pout = out;
      if ((Pin + 1) mod N ≠ Pout) break;
   }
   Copy the instruction trace pointers stored in
         the local buffer to the circular queue
         in the range Q[Pin, Pout − 1);
   Pin = (Pin + # of pointers copied) mod N;
   in = Pin;
   ...
   Generate instruction traces and store their
         pointers in the local buffer
   ...
}
```

(b)

**Figure 6.** The non-deterministic synchronization algorithm that accesses the circular queue without any synchronization. (a) The routine for FaCSim-acc (the consumer). (b) The routine for FaCSim-func (the producer).

In addition, the number of queue elements in range $Q[x, y)$ is denoted by $|Q[x, y)|$. The range of the traces that are valid in the queue for FaCSim-acc is $Q[\texttt{out}, \texttt{in})$, and for FaCSim-func is $Q[\texttt{in}, \texttt{out}-1)$. The queue is empty when $\texttt{in} = \texttt{out}$, and the queue is full when $(\texttt{in} + 1) \bmod N = \texttt{out}$. Note that the entry pointed by `in` is empty when the queue is full.

FaCSim-acc first takes a snapshot of the shared variables `in` and `out` and stores their values in the two local variables `Cin` and `Cout`. We assume that each assignment from the shared variable to the local variable or from the local variable to the shared variable is atomically performed. Then, `Cin` and `Cout` are used to determine whether the shared queue is empty in the while loop. If the queue is empty, FaCSim-acc stays in the while loop. Otherwise, FaCSim-acc exits the loop and copies the available pointers in the queue to its local buffer. When copying, the range of the available pointers for FaCSim-acc is determined by $Q[\texttt{Cout}, \texttt{Cin})$. After copying, FaCSim-acc sets `Cout` to the new location: `Cout = (Cout + # of pointers copied) mod N`, and commits `Cout` to the shared variable `out`.

The case of FaCSim-func is similar to FaCSim-acc. FaCSim-func takes a snapshot of the shared variables `in` and `out` into the two local variables `Pin` and `Pout`. Then, `Pin` and `Pout` are used to determine whether the shared queue is full in the while loop. If the queue is full, FaCSim-func stays in the while loop. Otherwise, FaCSim-func exits the loop and copies the generated trace pointers stored in its local buffer to the shared queue with the range $Q[\texttt{Pin}, \texttt{Pout}-1)$. After copying, FaCSim-func sets `Pin` to the new location: `Pin = (Pin + # of pointers copied) mod N`, and commits `Pin` to the shared variable `in`. The algorithm guarantees that FaCSim-func and FaCSim-acc always operate on the valid pointers.

| Versatile Application Baseboard, ARMulator, FaCSim | |
|---|---|
| Core | ARM9E-S, 5-stage, inorder, 140MHz |
| Data cache | write-back, no write-allocate,<br>32KB, 4way, 32B line, 1-cycle hit latency,<br>write buffer (8 data entries and one address entry) |
| Instruction cache | 32KB, 4way, 32B line, 1-cycle hit latency |
| Main memory latency (SDRAM) | 36-cycle non-sequential read (row hit)<br>48-cycle non-sequential read (row miss)<br>12-cycle non-sequential write (row hit)<br>30-cycle non-sequential write (row miss)<br>3-cycle sequential read (row hit, row miss)<br>3-cycle sequential write (row hit, row miss) |
| System bus | multi-layer AHB, 47MHz |
| Write buffer | 1-cycle latency, 16 entries with 4 addresses |
| Instruction prefetch buffer | 1-cycle latency, 4 entries |
| Data micro-TLB | 8-entries, 1-cycle hit latency |
| Instruction micro-TLB | 8-entries, 1-cycle hit latency |
| Unified TLB | 2-way 64 entries, 1-cycle hit latency |
| Compiler | armcc v1.2 without floating point unit |
| System call library | semihosting |
| **SimpleScalar** | |
| Core | StrongARM 1100, 5-stage, inorder,<br>floating point ALU |
| Data cache | 32KB, 4way, 32B line, 1-cycle hit latency,<br>16 entries with 4 addresses write buffer |
| Instruction cache | 32KB, 4way, 32B line, 1-cycle hit latency |
| Main memery latency | 12-cycle first chunk, 3-cycle inter chunk |
| Data TLB | 8-entries, 1-cycle hit latency |
| Instruction TLB | 8-entries, 1-cycle hit latency |
| Compiler | arm-linux-gcc v3.3.2 with floating point unit |
| System call library | linux system call emulation by SimpleScalar |

**Table 1.** Parameters of the simulated architectures. All cycles are 140MHz cycles.

## 4. Evaluation Environment

In this section, we describe our evaluation environment and applications used for the performance evaluation of FaCSim.

We compare FaCSim to the Versatile Application Baseboard (AB) for ARM926EJ-S[16] for its accuracy, and to RealView ARMulator ISS[17] and SimpleScalar[34] for its performance. Table 1 summarizes the architecture parameters of the development board and simulators. The architecture parameters of the Versatile AB, ARMulator, and FaCSim are the same. Their main memory latencies are obtained by measuring them on the Versatile AB and from the vendor's data sheet[35]. We configured SimpleScalar with the architecture parameters similar to those of the development board. It is almost impossible for us to configure SimpleScalar to have exactly the same architecture parameters as Versatile AB. The simulators are run on a host system with a 2.40GHz single Intel Core2Quad processor and 4GByte of physical memory. The processor has total four cores, and a pair of cores shares a 4MByte L2 cache. We use the embedded microprocessor benchmark applications from the EEMBC benchmark suite [5] for our evaluation. We have chosen 21 applications whose execution time is greater than 5 seconds on the Versatile AB. The execution time of each application on the Versatile AB is measured with the timer module on the board.

## 5. Experimental Results

When we compare the core cycles (i.e., core busy cycles) computed by FaCSim to that of ARMulator, the error margin is less than

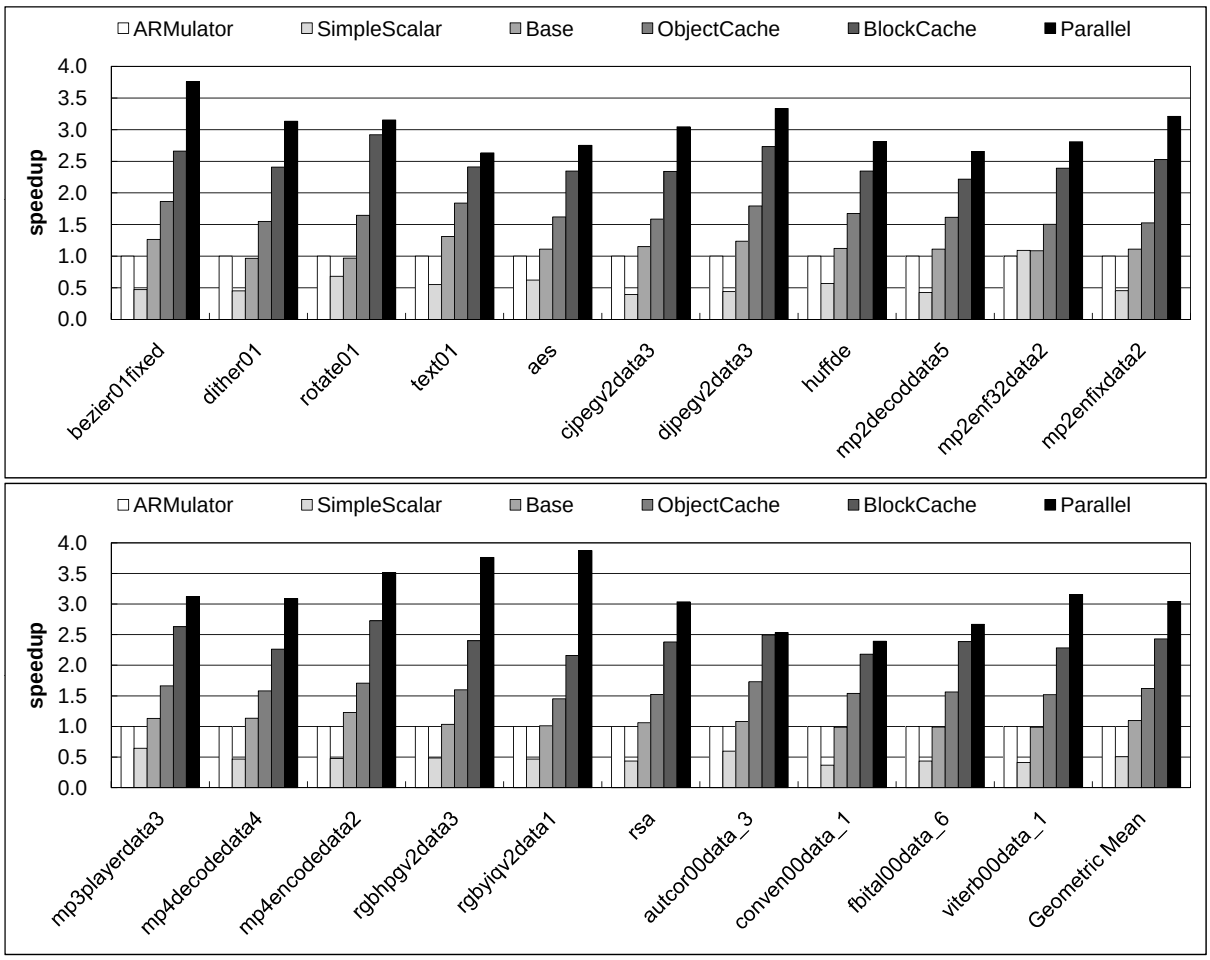


**Figure 7.** Speedup.

$\pm 1\%$. The documentation of ARMulator [17] claims that it is 100% cycle-accurate without instruction or data caches. For this reason, we can say that the core pipeline modeled in FaCSim is almost 100% accurate. Table 2 compares FaCSim against the ARM926EJ-S development board for their accuracy. We see that FaCSim is quite accurate, and the error margin is $\pm 6.79\%$.

Figure 7 shows the performance evaluation result of FaCSim for each application. It compares four different FaCSim implementations and SimpleScalar against ARMulator. The speedup is obtained over ARMulator. The four FaCSim implementations are FaCSim without any optimizations (Base), FaCSim with object caching (ObjectCache), FaCSim with basic-block caching (BlockCache), and parallelized FaCSim with basic-block caching (Parallel). Note that SimpleScalar is configured with a bit different but similar architecture parameters to those of the ARM926EJ-S development board.

Without any optimization (Base), FaCSim is on average 1.1 times faster than ARMulator, and on average 2.17 times faster than SimpleScalar. For mp2enf32data2, SimpleScalar beats ARMulator. The reason is that the applications compiled for SimpleScalar use floating point operations. That is not the case in ARMulator and FaCSim. By caching instruction objects (ObjectCache), FaCSim runs on average 1.62 times and 3.21 times faster than ARMulator and SimpleScalar, respectively. Caching instruction objects reduces the overhead of object creation and destruction. This significantly improves the performance. However, it still suffers from the cache lookup overhead. The basic-block caching technique removes this overhead significantly. With basic-block caching (BlockCache), it runs on average 2.43 and 4.81 times faster than ARMulator and SimpleScalar, respectively. After parallelization with the basic-block caching technique, FaCSim gets even faster. It is on average 3.04 times and 6.03 times faster than ARMulator and SimpleScalar, respectively.

The parallelized FaCSim has 1025 entries in the circular buffer. This is an optimal buffer size for a host system with a 4MB shared L2 cache between the two threads of FaCSim. We have found this value by experimentation. We performed sensitivity analysis of the buffer size on two cores with a 4MB shared L2 cache and on two cores each with a 4MB L2 cache. We varied the circular buffer size from 257 entries to 4097 entries and measured the number of cache misses with VTune analyzer[9]. For both L2 cache configurations, 1025 entries cause the smallest number of L2 cache misses. In addition, the cores with a shared L2 cache have about 100 times less L2 cache misses than the cores with separate L2 caches.

Table 3 summarizes the performance of the development board and the simulators in MIPS. The development board is about 18 times faster than FaCSim with a core clock frequency of 140MHz. Overall, the evaluation results imply that the techniques used in FaCSim for computing core clock cycles and improving performance are

| Image Name | Error |
|---|---|
| bezier01fixed | -4.10% |
| dither01 | -3.37% |
| rotate01 | -12.73% |
| text01 | 4.77% |
| aes | -9.99% |
| cjpegv2data3 | -5.83% |
| djpegv2data3 | -1.51% |
| huffde | -13.32% |
| mp2decoddata5 | -6.35% |
| mp2enf32data2 | -13.24% |
| mp2enfixdata2 | -5.44% |
| mp3playerdata3 | -10.21% |
| mp4decodedata4 | -0.43% |
| mp4encodedata2 | 0.12% |
| rgbhpgv2data3 | -2.85% |
| rgbyiqv2data1 | -2.42% |
| rsa | -6.43% |
| autcor00data_3 | 4.63% |
| conven00data_1 | -1.15% |
| fbital00data_6 | -3.55% |
| viterb00data_1 | -0.47% |
| **Root Mean Square** | **6.79%** |

**Table 2.** The accuracy of FaCSim when it is compared to the ARM926EJ-S development board.

| | Board | ARMulator | SimpleScalar | FacSim |
|---|---|---|---|---|
| MIPS | 77.10 | 1.43 | 0.80 | 4.25 |

**Table 3.** The performance of the simulators in MIPS.

effective and give quite accurate results for cycle-accurate simulation.

Currently, FaCSim with functionally modeled ARM peripherals, such as PrimeXSys System Controller (SP810), PrimeCell Vector Interrupt Controller (PL190), PrimeCell Multiport Memory Controller (GX175), PrimeCell Static Memory Controller (PL093), PrimeCell Timer Module(SP804), and PrimeCell UART (PL011), successfully runs an unmodified embedded Linux image (Linux-2.6.21-arm1).

## 6. Conclusion

FaCSim is intended for a fast and cycle-accurate micro-architectural simulation engine that can be used in full embedded system simulators. It is implemented in C++ from scratch. It demonstrates high-speed interpretive architecture simulation with cycle-level accuracy that is achieved by means of three mechanisms. First, it computes the elapsed cycles in each pipeline stage as a chunk and incrementally adds them up to advance the core clock instead of performing cycle-by-cycle simulation. Second, it uses a basic-block cache that caches decoded instruction objects at the basic-block level. This significantly reduces the overhead of creating and destroying objects. Finally, it is parallelized with a producer-consumer model to exploit tightly coupled multicore systems.

Experiments with 21 applications from the EEMBC benchmark suite validate FaCSim's accuracy with a $\pm 7\%$ error margin compared to the ARM926EJ-S development board from ARM. The result shows that FaCSim is, on average, more than three times faster than ARMulator and more than six times faster than SimpleScalar.

Currently, FaCSim with some functionally modeled ARM peripherals successfully runs an unmodified embedded Linux image. We believe that FaCSim is a first step towards a fast and cycle-accurate full system simulator, and it can contribute to the community for facilitating embedded architecture explorations, instrumenting and profiling embedded applications and operating systems, and performance debugging of embedded software in addition to functional debugging.

## Acknowledgement

We wish to thank Sangmin Seo and Jun Lee for their helpful comments on the preliminary version of this paper.

## References

[1] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2000. ACM Press.

[2] Martin Burtscher and Ilya Ganusov. Automatic synthesis of high-speed processor simulators. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 55–66, Washington, DC, USA, 2004. IEEE Computer Society.

[3] Derek Chiou, Dam Sunwoo, Joonsoo Kim, Nikhil A. Patil, William Reinhart, Darrel Eric Johnson, Jebediah Keefe, and Hari Angepat. FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 249–261, Washington, DC, USA, 2007. IEEE Computer Society.

[4] Bob Cmelik and David 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*, pages 128–137, New York, NY, USA, 1994. ACM Press.

[5] The Embedded Microprocessor Benchmark Consortium. EEMBC Benchmark Suite. `http://www.eembc.com`, 2008.

[6] James Donald and Margaret Martonosi. An Efficient, Practical Parallelization Methodology for Multicore Architecture Simulation. *IEEE Computer Architecture Letters*, 5(2):14–14, August 2006.

[7] Joel Emer, Pritpal Ahuja, Eric Borch, Artur Klauser, Chi-Keung Luk, Srilatha Manne, Shubhendu S. Mukherjee, Harish Patil, Steven Wallace, Nathan Binkert, Roger Espasa, and Toni Juan. Asim: A Performance Model Framework. *Computer*, 35(2):68–76, 2002.

[8] Lei Gao, Stefan Kraemer, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. A fast and generic hybrid simulation approach using c virtual machine. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 3–12, New York, NY, USA, 2007. ACM.

[9] Intel. VTune Performance Analyzer. `http://www.intel.com`, 2008.

[10] K. H. (Kane) Kim, Juan A. Colmenares, and Kee-Wook Rim. Efficient adaptations of the non-blocking buffer for event message communication. In *ISORC'07: Proceedings of the 10th IEEE15th International Symposium on Object and Component Oriented Real-Time Distributed Computing*, May 2007.

[11] Stefan Kraemer, Lei Gao, Jan Weinstock, Rainer Leupers, Gerd Ascheid, and Heinrich Meyr. Hysim: a fast simulation framework for embedded software development. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 75–80, New York, NY, USA, 2007. ACM.

[12] Gary Lauterbach. Accelerating architectural simulation by parallel execution of trace samples. Technical report, Mountain View, CA, USA, 1993.

[13] ARM Limited. *ARM926EJ-S Techinical Reference Manual*, 2003. `http://infocenter.arm.com`.

[14] ARM Limited. *ARM9E-S Core Techinical Reference Manual*, 2004. `http://infocenter.arm.com`.

[15] ARM Limited. *ARM Architecture Reference Manual*, 2005. `http://infocenter.arm.com`.

[16] ARM Limited. *Verstile Application Baseboard for ARM926EJ-S User Guide*, 2006. `http://infocenter.arm.com`.

[17] ARM Limited. *RealView ARMulator ISS User Guide, Version 1.4.3*, 2007. `http://infocenter.arm.com`.

[18] LISA - Language for Instruction Set Architecture. `http://www.iss.rwth-aachen.de/lisa/`, 2001.

[19] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[20] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Høallberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[21] Carl J. Mauer, Mark D. Hill, and David A. Wood. Full-system timing-first simulation. In *SIGMETRICS'02: Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 108–116, New York, NY, USA, June 2002. ACM.

[22] Christopher Mills, Stanley C. Ahalt, and Jim Fowler. Compiled instruction set simulation. *Software, Practice and Experience*, 21(8):877–889, 1991.

[23] Mayan Moudgill, John-David Wellman, and Jaime H. Moreno. Environment for PowerPC Microarchitecture Exploration. *IEEE Micro*, 19(3):15–25, May/Jun 1999.

[24] Achim Nohl, Gunnar Braun, Oliver Schliebusch, Rainer Leupers, Heinrich Meyr, and Andreas Hoffmann. A universal technique for fast and flexible instruction-set architecture simulation. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 22–27, New York, NY, USA, 2002. ACM Press.

[25] David A. Penry, Daniel Fay, David Hodgdon, Ryan Wells, Graham Schelle, David I. August, and Dan Connors. Exploiting Parallelism and Structure to Accelerate the Simulation of Chip Multi-processors. In *HPCA '06: Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 27–38, Feburary 2006.

[26] M. Poncino and Jianwen Zhu. Dynamosim: a trace-based dynamically compiled instruction set simulator. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 131–136, Washington, DC, USA, 2004. IEEE Computer Society.

[27] QEMU. `http://fabrice.bellard.free.fr/qemu/`, 2008.

[28] Wei Qin, Joseph D'Errico, and Xinping Zhu. A multiprocessing approach to accelerate retargetable and portable dynamic-compiled instruction-set simulation. In *CODES+ISSS '06: Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 193–198, New York, NY, USA, 2006. ACM Press.

[29] Mehrdad Reshadi, Prabhat Mishra, and Nikil 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*, pages 758–763, New York, NY, USA, 2003. ACM Press.

[30] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: The simos approach. *IEEE Parallel Distrib. Technol.*, 3(4):34–43, 1995.

[31] Eric Schnarr and James R. Larus. Fast out-of-order processor simulation using memoization. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 283–294, New York, NY, USA, 1998. ACM Press.

[32] Kevin Scott and Jack Davidson. Strata: A software dynamic translation infrastructure. Technical report, Charlottesville, VA, USA, 2001.

[33] SESC: SuperESCalar Simulator. `http://iacoma.cs.uiuc.edu/~paulsack/sescdoc/`, 2002.

[34] SimpleScalar. `http://www.simplescalar.com`, 2004.

[35] Infineon Technologies. *HYB39S512400T(L), HYB39S512800T(L), HYB39S512160T(L) 512-Mbit Synchronous DRAM Data Sheet, Rev. 1.3*, 2003. `http://www.infineon.com`.

[36] Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *SPAA '01: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, pages 134–143, New York, NY, USA, 2001. ACM.

[37] Steven Wallace and Kim Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 209–220, Washington, DC, USA, 2007. IEEE Computer Society.

[38] Emmett Witchel and Mendel 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*, pages 68–79, New York, NY, USA, 1996. ACM Press.

[39] Ji Zhang, Jaejin Lee, and Philip K. McKinley. Optimizing the java piped i/o stream library for performance. In *LCPC '02: Proceedings of the 15th International Workshop on Languages and Compilers for Parallel Computing*, pages 233–248, Berlin/Heidelberg, Germany, July 2002. Springer. Also published in Springer Lecture Notes in Computer Science, Vol. 2481/2005.

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