Ultra-Fast Functional Cache Modeling

Arianna Delsante
Abstract

Ultra-Fast Functional Cache Modeling

Arianna Delsante

Accurate cache and branch predictor simulation is a crucial factor when evaluating the performance and power consumption of a system. Usually, this is a complex and time-consuming task, therefore, simulators tend not to model any cache system by default due to practical constraints. This thesis project proposes a new way to collect data for the simulation of cache and branch predictor devices whose speedup is of orders of magnitude compared to existing Simics models while maintaining the same level of accuracy. A subset of benchmarks from SPEC CPU 2006 suite is used to investigate the effect of different data collection methods on the simulation’s slowdown.

The Simics simulator framework is heavily used by Intel and is used in this thesis work. It is a modular full-system simulator capable of simulating an entire system while running on the user developing environment. The newly developed Simics instrumentation framework, its API and the just-in-time (JIT) compiling technology are used and expanded. The aim is to implement new ways of collecting and sharing data between the new models and the simulated processor for minimum slowdown when computing statistics for cache and branch predictor models.

The implemented cache model is on average 30 times faster and up to 40 times faster when compared to a simple but well tested and tuned sample cache model shipped with Simics. The same performance comparison can't be done for the newly modeled branch predictor since Simics currently does not model any branch predictor, however, it introduces minimum overhead, 1.4 on average, when compared to Simics running without any extension model.

This new mechanism of collecting data for cache and branch predictor simulation makes it possible to run realistic workloads and at the same time collect cache and branch predictor statistics for live analysis while maintaining interactive user experience and overall keeping the simulation's slowdown extremely low.
# Contents

1 Introduction .................................................. 1
   1.1 Fast Full-System Simulation ................................. 1
   1.2 Project Objective ........................................... 2
   1.3 Structure of the Report ...................................... 2

2 Background ................................................... 3
   2.1 Simics Overview ............................................. 3

3 Test Methodology and Configuration ..................... 10

4 Cache Model Implementation .............................. 11
   4.1 Cache Model Specifics ....................................... 11
   4.2 Cache Model using Timing-Model Interface ............... 12
   4.3 Cache model with Instrumentation API .................... 12
   4.4 Callback Overhead Analysis ................................ 14
   4.5 Optimization ............................................... 17
      4.5.1 Line Buffer (L0 cache) ............................... 17
      4.5.2 Batch N accesses for late execution ................. 19
      4.5.3 Line buffer, Batched data access, Events ......... 20
   4.6 Data and Instruction Cache ................................ 20
   4.7 Full model performance analysis ......................... 22

5 Branch Predictor Implementation ....................... 25
   5.1 Model Specification ......................................... 25
   5.2 Branch Predictor implemented with the Instrumentation API . 25
   5.3 Buffer and Event Implementation ......................... 27
   5.4 Performance Results ...................................... 28

6 Future Work ................................................ 30

7 Conclusion .................................................. 31
# List of Figures

2.1 Simics Architecture ................................................. 4  
2.2 Simics Simulator API examples. ................................. 5  
2.3 Simics objects and interfaces ................................. 6  
2.4 Transaction Path through Simics Memory[8] ............... 7  
2.5 A simple tool using with the instrumentation framework. 7  
2.6 Instruction execution in Simics[2] .............................. 8  
4.1 Cache model architecture when implemented using the timing-model interface. ................................. 12  
4.2 Slowdown of the cache model implemented with the timing model interface, lower is better. The execution time is normalized to Simics running without caches enabled in JIT mode. ................................. 13  
4.3 Cache model implemented with the instrumentation API ................................. 13  
4.4 Slowdown comparison between a timing model implementation and the implementation with the new instrumentation API. The slowdown is normalized to Simics running without caches enabled in JIT mode. 14  
4.5 JIT Callout ....................................................... 15  
4.6 Percentage of slowdown due to callbacks with no statistics computed in the model. The simulation is normalized to Simics running without caches enabled in JIT mode. ................................. 15  
4.7 Slowdown study ..................................................... 16  
4.8 Callback average overhead analysis when running *astar* from SPECint 2006. ................................. 16  
4.9 JIT code L0 optimization in the JIT code function *Read_Before*. ................................. 18  
4.10 Optimization with a common line buffer in JIT code for read and write access. The execution time is normalized to Simics running without caches enabled in JIT mode. ................................. 18  
4.11 Optimization with 10 000 batched instructions and a common line buffer for read and write. The execution time is normalized to Simics running without caches enabled in JIT mode. ................................. 19
4.12 Slowdown comparison between the cache model implemented with instrumentation callbacks and event posted by the model. The execution time is normalized to Simics running without caches enabled in JIT mode. ................................. 21
4.13 Address bits. .................................................. 21
4.14 Cache model with data and instruction cache. ................. 22
4.15 Full model design. .......................................... 23
4.16 JIT code Call_Instrumentation_Read_Before in .md file. ....... 24
4.17 Cache model performance slowdown when running astar from SPECint 2006, where the execution time is normalized to Simics running without caches enabled in JIT mode. ................................. 24

5.1 Branch predictor gshare scheme [12]. .......................... 26
5.2 Branch predictor model design. .................................. 26
5.3 Branch predictor overhead with callbacks for every branch instruction. The execution time is normalized to Simics running without any extension model. .......................... 27
5.4 Instrumentation_After primitive in JIT code, without optimization. . 27
5.5 Instrumentation_After primitive in JIT code with optimization. . . 28
5.6 Branch predictor implementation make_prediction function. ....... 28
5.7 Branch predictor slowdown comparison between the implementation with a callback for each branch address and batched instructions. The execution time is normalized to Simics running without any extension model. ................................. 29
Chapter 1

Introduction

1.1 Fast Full-System Simulation

Modern computer systems are very complex in their nature where both hardware and software need to be optimized carefully to fulfill all requirements of performance, low power, and security. This increases the demand for modeling techniques that provide both accuracy and high simulation speed to explore various design approaches. A higher level of detail in the simulation can help designers to effectively choose different algorithms. Simulations can decouple software development from hardware availability which is important for reducing time to market. Moreover, software construction benefits from simulation environments where the modeled system can be inspected and controlled while extracting measurements that are deterministic and non-intrusive[8].

One of the key aspects of a simulation model is the trade-off between simulation performance and model accuracy. While ideally, the simulated system aims to be a perfect model, it must be sufficiently abstract to allow satisfactory performance levels. Generally, this is done by providing the system with sufficient functional and timing accuracy to model realistic workloads. Functional accuracy refers to how well the behavior of the system is simulated, timing accuracy refers to the similarity between virtual time when compared to the time of a real system running the same software.

To accurately model a processor, the simulator should also model different hardware features whose goal is to increase CPU performance. For example, the behavior of a cache[11] where hit and miss rates can have dramatic effects on a machine performance as well as different algorithms for branch predictors that allow the processor to resolve the outcome of a branch preventing stalls[7]. However, simulations become significantly time costly as accuracy increases.

Simics is one of the simulation tools used by Intel Corporation, it is a modular full-system simulator capable of simulating an entire system. Simics is a fast, scalable, flexible and extendable framework that can simulate large and complex systems consisting of a broad range of different hardware models including CPUs, memories, devices and even complete networks of machines. Simics does not model any cache system by default since collecting data for cache and branch predictor simulation would significantly slow down the simulation. Such data are usually not needed when
ensuring software correctness, but important when evaluating the performance and power consumption of the system.

1.2 Project Objective

This project analyzes the performance of a provided simple but well-tuned sample cache implementation shipped with Simics and proposes a new way of collecting data for detailed cache and branch predictor simulation on modern processors with minimal overhead. Simics is already provided with cache hierarchy models that can be added as extensions when running a simulation, however, to get hold of the necessary information for the current model slows down the simulation significantly for its implementation requires certain Simics features e.g. the Simulator Translation Cache (STC) and JIT compiler to be switched off when the model is attached. The instrumentation framework is a new feature available in Simics that provides a mechanism for inspecting various part of the simulated system in a flexible way allowing the collection of the necessary data for cache simulation without turning off any other Simics feature needed for performance (the instrumentation framework will be further explored in the next chapters).

This project’s innovation, therefore, is not located in the state of art of cache organization such as replacement policies but looks into new ways of developing ultra-fast simulation models for both caches and branch predictors. The new framework is used and extended, new features are added to collect the necessary information for cache and branch predictor simulation in a fast way. The newly developed interfaces in the internal simulation engine, including the just-in-time (JIT) compiler, is used and new interfaces are implemented for an ad hoc communication between the developed models and the simulated processor. The outcome is expected to be a new model device that is many times faster compared to the existing simple model shipped with Simics while maintaining the same level of accuracy.

1.3 Structure of the Report

This section provides a structural outline of the thesis. After a brief explanation of some of the terms used in this paper, Chapter 2 starts out by introducing a basic understanding of Simics and its features. Chapter 3 provides information about the configuration of the system and test methodology in use. Chapter 4 follows with a detailed explanation of the cache model optimization process. It starts out with an analysis of the available model in Simics, the model implemented with the new instrumentation API, its pitfalls and the optimization trade-off to achieve maximum speedup. The branch predictor implementation and the resulting performance is explained and evaluated in Chapter 5. Chapter 6 is concerned with the discussion, including possible applications and future work.
Chapter 2

Background

2.1 Simics Overview

Simics [1] is an instruction set simulator. The native level of timing abstraction is software timing: time passes as instructions are being executed and all memory accesses complete in zero time. The smallest visible unit of execution is a single instruction, although different parts of instruction like memory accesses can be observed through interfaces. By default a simulated instruction takes one simulated cycle to execute - the default can be changed by configuration or extended to simulate detailed timing. Simics does not model internal hardware components like instruction pipelines, buffers, and memory controllers. However, this simple approach is usually enough to run any production code without adjustments.

Simics is modular, it provides a small simulation core, an API and other functionality to include different models at runtime. Extensions are loaded from Simics modules. The whole system can be updated and upgraded independently, it is possible to adjust the simulation system while running, e.g. adding features or re-configuring the target hardware and it is possible to add more levels of detail, i.e. slow down execution for memory operations.

Simics runs on the simulation host. Inside a Simics process, there are one or more simulation targets including the software of a target platform. More details about Simics architecture are shown in Figure 2.1. Simics models the basic behavior of fast functional processor that can run any software. The target hardware is what Simics simulates, on top of this the target software is running.

The simulated target system also includes device models as shown in Figure 2.1. Device models are passive and reactive, they wait to be activated by the processor or external events. Operations are atomic and model ”what” the device does and not ”how” it does it.

Simics device models can be written using different languages: DML\(^1\), Python and C/C++. In this project the prevalent programming language used is C. It can be used for writing any type of Simics module, it exposes the module to all interaction in Simics and it is fast. Infrastructure and features in Simics include for example

\(^{1}\)DML (Device Modeling Language) is a programming language for modeling devices in Simics. DML has been designed to make it easy to represent the kind of things that are needed by a device model, and uses special syntactic constructs to describe common elements such as memory mapped hardware registers and connections to other Simics configuration objects.
real-world connections, the new instrumentation used in this project, breakpoints and debugging features.

Simics APIs are designed to allow the integration of new modules in the Simics family of products. They are divided into three major groups and have three additional groups of interfaces. An interface is a set of methods that allow objects to interact with each other. The three main groups are:

- **Device API**: includes the basic functionality of the Simics API and the only part available to device models.
- **Simulator API**: contains the complete Simics API, including parts not available to models.
- **Processor API**: extends the Device API with functions used when modeling processors in Simics.

In this project, the Simulator API is used. Figure 2.2 shows an example of its use. The `SIM_register_interface` is used to register the interface for a specific class, in this case the `conn_class`. The function `SIM_register_event` is used to register an event for the same previous class so that it can be posted to interrupt the simulation after a defined number of steps.

Objects and their attributes describe the configuration of Simics. Devices are objects that use attributes to carry their state. Modules in Simics define classes and from those classes, objects are instantiated and connected to the system configuration as viewed in Figure 2.3.

The attributes of an object are a fundamental concept in Simics, they are used to:

- define parameters and configuration,
- reference to other objects,
- carry the current simulation state.
2.1. Simics Overview

SIM_register_interface:

```c
static const connection_interface t ct iface = {
    .register_parameters = register_parameters,
};
SIM_register_interface(conn_class, CONNECTION_INTERFACE, &ct iface);
```

SIM_register_event:

```c
event = SIM_register_event("event", conn_class, 0,
    start_simulation,
    NULL, NULL, NULL, NULL);
```

Figure 2.2: Simics Simulator API examples.

Objects use interfaces to interact with each other. Interfaces define different characteristics of the objects. They have names and are made of a list of functions defined by each object’s class implementing the interface. An interface does not include any data.

The simulation in Simics is event-based, it is driven by executing a sequence of events. An event is a specific change at a specific point in virtual time. Executing an event has the consequence of changing the state of the simulation. A simulation progresses as a sequence of such state transitions. As Simics makes the simulated time progress, cycle by cycle, events are triggered and executed. Events include device interrupts, internal state updates, as well as step executions. A step is a unit in which Simics divides the execution of a flow of instructions going through a processor; it is defined as the execution of an instruction, an instruction resulting in an exception, or an interrupt. All the actions are taken as a result of events, once the event is handled, control is returned to the simulation kernel. In Simics, events are scheduled on event queues typically attached to processor models[8]. A module can react to specific Simics events, e.g. processor exceptions or control register writes.

Memory Simulation Modules and Interfaces

In Simics, each memory subsystem is simulated by an object. When a simulated processor executes a load or store to an address, Simics checks for memory spaces that provide support for generic 64-bit address spaces into which memory and devices can be mapped. A memory space takes a stream of transactions to address space and distributes them to devices mapped into the address space in a highly efficient manner. Simics memory-spaces are handled by the generic memory-space class. A memory-space object implements interface functions for memory accesses, and it has attributes specifying how the mappings are set up.

A logical address is an address used by a program, sometimes called a virtual address. On the target architecture, this address is translated to the physical address by the memory management unit (MMU). The physical address is then used to look up the data. Since the simulator itself exists in a virtual-physical environment, the physical address needs to be translated to the real address, an address in the simulator’s virtual address space.

Memory spaces provide a memory hierarchy interface for observing and modify memory transactions passing through them. The interface is composed of two dif-
Ferent interfaces acting at different phases of a memory transaction execution. The one used by the current model shipped with Simics is the timing model interface: it provides access to a transaction before it has been executed. An object can be connected to this interface by registering the value of the object as the timing model attribute of the corresponding memory space.

As shown in Figure 2.4 when the CPU loads an instruction, a memory transaction is created. If the address is in the STC, the data is read and returned to the CPU using the cached information. If the address is not in the STC, the transaction reaches the CPU memory space. If a timing-model is connected to the memory space, it receives the transaction. The memory space determines the target object (in this example a RAM object). If possible the transaction is inserted in the STC. The limitation of the current model already shipped with Simics is the use of the timing model interface to get hold of the addresses for cache simulation. In fact, to get correct statistics the STC needs to be disabled and the JIT engine can not be used.

Instrumentation Framework

The instrumentation framework provides a mechanism to inspect various parts of the simulated system while running. An instrumentation provider is an object that can supply information on things happening in the system. Instrumentation tools are objects that can be connected to providers and subscribe for information by registering for specific callbacks. Instrumentation connections are communications channels that are used when a tool and a provider are connected.

For example, it is possible to register a callback for every data access being performed by the processor, the tool can extract the address with the use of the instrumentation
2.1. Simics Overview

Figure 2.4: Transaction Path through Simics Memory[8]

Figure 2.5: A simple tool using with the instrumentation framework.

API and use it to model the behavior of the cache model. With the use of the instrumentation API the cache model does not require data accesses to reach the memory space, instead, it registers for specific callbacks directly to the processor. The callbacks are invoked every time there is a data access. A tool can also filter the instructions that will cause a callback. The latter is used for example in the implementation of a branch predictor model, see Chapter 5, so that the model will be called whenever specific instructions are about to be executed, such as conditional branches.

Simics runtime and modes of execution

When running in a fast virtual platform like Simics most of the time is spent running instructions. As shown in Figure 2.6 Simics can run in interpreter mode, use just-in-time technology to convert target code to host code or run code using the Intel Virtualization Technology (Intel VT-x) known as VMP.

JIT (just-in-time) systems are used to improve the time and space efficiency of a program by combining the benefits of both (static) compilation and interpretation. Dynamic compilation refers to a translation that occurs after a program begins
Simics VMP technology makes it possible to directly execute target code on the host processor thus get close to native speed. In this context, however, this optimization is not used because it does not support the previously mentioned instrumentation when enabled.

JIT operates as an add-on to the plain interpreter. The interpreter mode is rarely used nowadays. Instead, JIT techniques are used to make the slowdown of the virtual machine as low as 1x when running code[3]. JIT compiler translates target instructions into native code that runs on the host, it has a static and dynamic part: the static part is used to compile target instruction definitions into templates when Simics compiles, the dynamic part joins templates into blocks at run-time. A block is a single entry built from instruction templates which code is stored in an internal representation inside Simics; at run-time, the templates for instructions considered important enough, that is, performed a relevant amount of time, are merged in a block and the entire block is further optimized. JIT also holds the code for handling instrumentation calls to C code. This was not available before the introduction of the instrumentation framework, in fact, it was not possible to register callbacks to get hold of specific information, e.g., memory accesses, from the JIT engine.

Figure 2.7 wraps up Simics architecture and how the JIT compiler and interpreter are placed inside the target machine.
2.1. Simics Overview

Figure 2.7: Simics: JIT compiler and interpreter[5]
Chapter 3

Test Metodology and Configuration

System Configuration

The hardware in use for the project is an X86 architecture running in 64-bit mode. It is provided with CPU running 8 hardware threads. Model 60 and model name Intel(R) Xeon(R) CPU E3-1245 v3 @ 3.40GHz. The target system in use is Yocto 1.8 core-image-full-cmdline with Linux 3.14 for PC BIOS with an X86 Nehalem (First Core generation) architecture.

Simics version

The version of Simics in use is Simics 5.

Test workloads

Performance is the main focus in the project. A set of benchmarks provided by SPEC (Standard Performance Evaluation Corporation) is used to evaluate the model performance. The interest in using SPEC benchmarks comes from their being based on real applications [6].

A total of 12 different benchmarks are run for various tests, each of them chosen so that different workloads and cache misses are tested. During the studying process only a subset of benchmarks is run. For more information about the benchmarks please refer to SPEC CPU 2006 publication [6].
Chapter 4

Cache Model Implementation

In this chapter are presented the different steps taken to develop an ultra-fast functional cache model. It starts by introducing the specifics of the model that remain constant throughout the different implementations. The cache model implemented using the timing model interface is introduced in Section 4.2. This way of implementing a cache model is the same one used in the already existing cache model in Simics. The reason behind this is to have a baseline for measurements of the speedup of the model under development. Section 4.3 presents the model implemented using the new instrumentation API available in Simics and how the slowdown of collecting data for accurate simulations is reduced by the new implementation. The model implemented with the new instrumentation API is then inspected for further optimization, this is done in Section 4.4 where the overhead is decoupled in its components. The results of this section are the starting point for an ad hoc optimization of the model in Section 4.5 where various ways for optimization are explored.

After the model has been optimized, its scalability to more accurate models is tested. Firstly, in Section 4.6 the model is presented extended with an instruction cache then, in Section 4.7, are analyzed the performance of the model with both first-level instruction and data cache.

4.1 Cache Model Specifics

The cache model is 64 KiB and direct-mapped, therefore, no replacement policy is implemented. The reason behind this is to analyze the slowdown of a simplistic model to understand the cost of collecting the right data for the simulation of more realistic models.

The model can be customized according to user needs, i.e. cache line size, levels of set associativity and size. It is implemented as a separate module from the simulated system and can be included at runtime.

A subset of benchmarks from SPECint and SPECct CPU 2006 is used to measure the model’s performance.
4.2 Cache Model using Timing-Model Interface

The model implemented with the timing model interface uses memory-space objects (2.1) to observe memory transactions passing through them. They provide a memory hierarchy interface that gives access to a transaction before a memory operation is executed. The object connects to the memory space of the simulation through the timing model interface by setting the timing model attribute of the memory space to the object that is going to be connected. In this way, whenever memory access is performed, the object will be called see Figure 4.1. This was the only way to get hold of memory operations before the introduction of the instrumentation API that is used and extended in this work.

![Cache model architecture when implemented using the timing-model interface.](image)

Figure 4.1: Cache model architecture when implemented using the timing-model interface.

The implementation aims to provide comparison values for speedup studies and behavioral information of the new model. As shown in Figure 4.2 the overhead of running with a simple cache model is very high when using the timing-model interface.

For performance reasons Simics generally does not perform all memory accesses through memory spaces instead, the Simulator Translation Cache (STC) caches relevant information[9]. When fetching an address from the STC no memory access is performed, therefore, when an object is listening for specific transactions on the timing model interface, the STC is disabled to provide all the memory accesses to the cache simulator. This slows down the simulation significantly.

The aim of this project is not to optimize the currently available model but to work towards a new solution that uses the new available instrumentation API for developing a new way of collecting the data needed for the model’s simulation.

4.3 Cache model with Instrumentation API

Design

The model is implemented as an instrumentation tool that subscribes to data accesses information to the processor. The cache module is loaded on the system and
4.3. Cache model with Instrumentation API

Figure 4.2: Slowdown of the cache model implemented with the timing model interface, lower is better. The execution time is normalized to Simics running without caches enabled in JIT mode.

![Graph showing slowdown comparison]

Figure 4.3: Cache model implemented with the instrumentation API

A new object is created and connected to the processor. The model registers processor interfaces from which it obtains information about each data access. It is a flexible implementation so that it can be extended with multiple cache levels.

When a new object is allocated, callbacks are registered in the connection object for every explicit, i.e., normal loads and stores, or implicit data access, such as table walks or memory accesses performed by exceptions and interrupt handling. A callback is handled by two different functions depending on the nature of the access, read or write.

Performance comparison between the cache model implemented with the timing model interface and the implementation with the instrumentation API is showed in 4.4.

In figure 4.2 the cache model using the timing model interface is extremely slow compared to the implementation with the instrumentation API, therefore, the maximum value of the y-axis has been set to 40 in order to fit both models within the
Chapter 4. Cache Model Implementation

Figure 4.4: Slowdown comparison between a timing model implementation and the implementation with the new instrumentation API. The slowdown is normalized to Simics running without caches enabled in JIT mode.

same chart, the complete picture of the slowdown is shown in Figure 4.2. The large "slowdown" in the cache model running with the timing-model interface is due to the simulation running in stall mode. In this case, the STC is not used because the memory accesses need to reach the memory space object for the model to be called, as described in Chapter 2. Currently, the STC caches 4K pages, therefore, it could not be used for cache simulation because it has different granularity. Since the size of the STC pages is not configurable in JIT mode, to use the STC for cache simulation would require the simulation to run entirely in interpreter mode which would give no performance. Instead, when running with the instrumentation API, the simulation can still use the STC since the objects are not listening on the timing model interface for data accesses but they are directly called by the processor before a data access occurs. This is one of the main advantages of using the instrumentation framework.

4.4 Callback Overhead Analysis

This section analyzes the overhead introduced by a callback when the code that models the cache is called for every data access. When the cache model is attached to the simulation it registers for a callback to the processor for read and write data accesses through the instrumentation API. The simulation would then run specific routines for every data access so that the cache model is called for each of them, both from JIT and the interpreter. When a data access is performed Simics either interrupts a JIT block to call the model from normal C code or, if already in the interpreter path, performs a call to the model directly. The slowdown due to calling the model and compute cache statistics are analyzed. To implement accurate and effective optimization it is important to understand where the most expensive parts of the simulation are.

In Figure 4.6 the total slowdown is decoupled between the one due to the cache simulating statistics and the one solely due to system calls to the model. The empty
4.4. Callback Overhead Analysis

Figure 4.5: JIT Callout

callback bars show the slowdown of the simulation when the cache model receives callbacks but does not update the statistics.

Figure 4.6: Percentage of slowdown due to callbacks with no statistics computed in the model. The simulation is normalized to Simics running with no caches enabled in JIT mode.

The figure shows that the slowdown introduced by receiving a callback for each data access amounts to 68% on average. The nature of the callback’s overhead is then analyzed to address the slowdown more accurately.

Firstly the slowdown exiting JIT blocks to reach the C code is analyzed. That is, the overhead caused by recreating the simulated PC since it is not maintained in the JIT block and recreate the number of elapsed cycles. Various studies are done to understand the most expensive parts of performing a callback:

- **Empty callbacks:** refer to the simulation when the cache model is reached by the callback but no statistics are computed (paths 1, 2 and 3 in Figure 4.7). It shows the overhead of collecting the necessary data for the model to simulate accurate statistics.

- **Exit JIT block:** is the overhead due to a JIT block being interrupted (path 1 in Figure 4.7), in this case, nothing is computed in the callout. It returns
right after the block is exited. The overhead is the cost of doing a call to the C code, setting up the stack frame for the callout to adhere to the ABI.

- **JIT callout**: refers to functions called from JIT into the C code (path 2 in Figure 4.7). The function handling the callout needs to get additional information to compute the correct program counter, therefore, the overhead is partly caused by the program counter being synchronized with the CPU structure as well as creating the callback info accessible from the model, e.g. physical address, virtual address, instruction length.

- **Interpreter overhead**: refers to the callback being issued from functions in the interpreter code (path 3 in Figure 4.7). Where memory information and memory operations are handled before reaching the simulated cache.

The analysis is done running *astar* from SPECint 2006. The average slowdown on multiple runs is 60%. The most expensive part of the callback is to go from JITed code to the callout in the C code as viewed in Figure 4.8. In this scenario two optimizations are proposed: a line-buffer tracked by
the processor (L0 cache) in Section 4.5.1, this would significantly reduce the overhead introduced by the JIT engine when calling the code for the simulated cache model, and a buffer storing each data access for late processing presented in Section 4.5.2, this would coalesce the calls to the cache model in one at the cost of filling the buffer with each address.

4.5 Optimization

4.5.1 Line Buffer (L0 cache)

The first idea for speeding up the system is to implement a line buffer that stores the latest data access written in the cache. The line buffer is stored in the processor structure, it is configured when the cache extension is attached to the simulation and it is accessible without any call to the model. If the two addresses are mapped to the same cache line the simulation does not reach out to the cache model, instead, one of the counters keeping track of the total number of read/write accesses is incremented. If the address does not belong to the same cache line, the line buffer is updated with the latest address, both from JIT code and the interpreter code. In this case, the percentage of JIT callout is reduced in case of high data locality; the more subsequent addresses belong to the same cache block the less callout will be performed. When the extension is loaded the block size is used to compute the offset bits of the addresses.

Implementation

When the simulation goes through the interpreter path a callback will always reach the model. This optimization is limited to JITted code since, as shown in Figure 4.8, that is the most expensive part of the simulation. The interpreter code only updates the line buffer with the latest address being written or read in the cache. The optimization is implemented using machine definition language, primitives handling `read_before` and `write_before` callback are modified. The CPU has two pointers to the counters storing the total number of read and write accesses in the model, those are incremented when a new address hits the line buffer. In this scenario, no translation to physical address is needed since the virtual address is used to check an L0 hit. To use the virtual address is an approximation for performance reasons with the assumption that it is very unlikely that two consequent memory operations use the same virtual address to map different physical addresses.

Performance Analysis

The presence of a line buffer reduces the percentage of callbacks of 30% on average. Performance gain is showed in Figure 4.10. This solution is faster and adaptable even to a multi-processor system since every processor has its interface to the model and each of them can configure a different size of the line buffer.

One approach for further speed up could be an additional line buffer in the processor, however, this is not the chosen approach since it may cause problems in terms of adaptability and scalability. Adding multiple lines would eventually cause problems
Figure 4.9: JIT code L0 optimization in the JIT code function `Read_Before`.

when the implemented cache has a different replacement policy. The aim is to implement a larger scale optimization.

Figure 4.10: Optimization with a common line buffer in JIT code for read and write access. The execution time is normalized to Simics running without caches enabled in JIT mode.
4.5.2 Batch N accesses for late execution

A second approach to reducing the number of calls to the cache model is to batch N data accesses in a buffer and perform a late computation of the statistics. This is expected to reduce the slowdown introduced by both JIT and interpreter callout since instead of being performed for each access this is done every Nth data access. However, new instructions need to be added both in JIT and interpreter code. From the JIT engine side, the address that is stored in the buffer needs to be converted from virtual to physical address, this is done by getting hold of the page address and the address offset stored in the STC thus causing little overhead. The buffer needs to be filled and the pointer moved to the next location, the counter keeping track of the number of addresses written into the buffer is incremented each time.

The weight of the optimization is slightly reduced by the lines of code added in the JIT code file.

The n-th data access causes a call to the model where the buffer is read out and the pointer moved back to the start. This implementation works in parallel with the line buffer implementation. If an address belongs to the same cache block as the previous address, the new address is not placed in the buffer instead, the number of reads or write accesses is incremented. The optimization is implemented both in JIT and the interpreter code, therefore, no callback is received until a certain number of accesses is triggered.

![Figure 4.11: Optimization with 10,000 batched instructions and a common line buffer for read and write. The execution time is normalized to Simics running without caches enabled in JIT mode.](image)

Performance Analysis

This implementation results in a significant speedup reducing considerably the amount of performed callbacks as shown in Figure 4.11. The performance gain is explored by running various benchmarks from SPECint and SPECft, results are normalized to the simulation without any model attached. The "speedup" of the model implemented with a buffer batching 10k instructions is lower than expected: theoretically,
we could have expected a reduction of 10000x in the overhead of calling out to C code. This is probably due to the changes made in JIT code for batching instructions and the line buffer. The current model compromises on performance because it requires the following implementations both in the JIT code and interpreter code:

- Shifting and comparing the current address being accessed and the one stored in the line buffer, and possibly update the line buffer with the new address,
- Computing the physical address from the virtual address for each data access,
- Encoding the nature of the access (read/write) in the address,
- A counter keeping track of the number of addresses written in the buffer that is increased and checked for each access,
- A \textit{jump} instruction for either writing the address in the buffer or, if full, call the cache model.

As a result of these points, the model does not fulfill the theoretical performance expectations, however, the model clearly outperforms the previous one.

\textbf{Statistics Granularity}

The model's statistics are not up to date with the current state of the system since the addresses are kept in the buffer until the buffer is filled. However, if this might seem like a limitation, in fact, if the simulation stops at anytime the statistics are always up to date with the simulation. This is because a special callback called \textit{Core_Simulation_Stopped} is registered in the model so that whenever the simulation goes back to the prompt the buffer is emptied from all the addresses and statistics are updated.

\textbf{4.5.3 Line buffer, Batched data access, Events}

During the simulation, the number of addresses in the buffer is tracked with a counter placed both in the JITed code and the interpreter. When filling the buffer with data accesses the counter keeping track of the number of addresses added in the buffer is checked and updated for each memory operation. The idea of posting an event by the cache model every N steps is to solely use the internal time in Simics that keeps track of the number of steps in the simulation thus removing a \textit{compare}, a \textit{jump} and an \textit{increment} instruction for each data access. The buffer dimension is designed big enough so that it will never overflow.

The cache model posts an event every 10 thousand steps.

\textbf{4.6 Data and Instruction Cache}

A new cache object for simulating instruction cache statistics is added. This way it is possible to obtain performance information of a more realistic model and evaluate how the implemented model can scale up when adding functionality.
4.6. Data and Instruction Cache

Figure 4.12: Slowdown comparison between the cache model implemented with instrumentation callbacks and event posted by the model. The execution time is normalized to Simics running without caches enabled in JIT mode.

Implementation

To proceed with the implementation of an additional instruction cache, a new object class is created for modeling both data and instruction first-level cache. The connection reads out the addresses from the buffer and works as a splitter to send the respective data and instruction addresses to the first-level caches.

The last two bits of the offset are modified according to the nature of the access: the least significant bit differentiates an instruction being fetched from data access. The second least significant bit differentiates read or write accesses. This is possible since the least significant bits are not part of the cache address so they can be modified to encode address information as shown in 4.13. This sets the boundaries for a cache line not to be smaller than 4 bytes.

A line buffer is added for instruction fetched mapped to the same cache block as previously implemented for the data cache. Both instructions and data accesses are batched in the same buffer to keep the order. Once the buffer is read out in the connection, addresses are sent to the first-level cache.
4.7 Full model performance analysis

Performance studies of the full model aim to analyze how it scales up when adding functionality. The structure of the model is shown in Figure 4.15 where two more cache levels are added as new objects from the same class.

Figure 4.16 compares JIT code before and after the model is optimized. As shown in the optimized version of the figure, at first we need to load both the address that is being read and the address that is stored in the line buffer in two registers. The addresses are then shifted and compared to check if they belong to the same cache line. If that is the case we increment the counter keeping track of the total number of read data and we jump to the end. If the two addresses do not belong to the same cache line we need to update the line buffer with the new address, get hold of the current physical address and write it in the buffer.

One of the main differences between the old and the new version is the presence of a callout into C code. This is replaced in the new version with the buffer storing each physical address not belonging to the same cache line.

The model’s performance is tested when adding more functionality not only in terms of new cache levels being added but also in terms of replacement policies. For this reason, an LRU replacement policy is added to the model. In Figure 4.17 is showed how this affects performance:

- The first bar shows the slowdown introduced by filling the buffer with addresses and trigger the readout with an event without computing the statistics in the model.
- The second bar shows the slowdown when the model has both first-level instruction and data cache, 2 ways set associative.
- The third bar shows the slowdown of the model when the first level caches are 8-way set associative.
The fourth bar shows the slowdown introduced by the model when both a second and a third level cache are added to the model. In this setting, all the objects in the hierarchy are 8-way set associative.

Figure 4.17 shows the overhead of the cache model when adding functionality. The benchmark *astar* is chosen as representative in the set of benchmarks from SPECint 2006. The overhead in the model increases when increasing the associativity of the L1 caches. When the model runs with an L2 and L3 cache the overhead increases despite the high hit rate in the L1 cache. This is partly caused by the LRU replacement policy implementation which is not optimized for caches much larger than 64K. The size of the L2 and L3 cache is greater than an L1 cache, therefore, even though the number of accesses is low, the slowdown of both writing back the addresses from the previous level and fetching the addresses has a higher impact on the model performance.
Old version:

```
Chapter 4. Cache Model Implementation

Old version:

```

Optimized version:

```

Figure 4.16: JIT code Call_Instrumentation_Read_Before in .md file.

Figure 4.17: Cache model performance slowdown when running *a*star from SPECint 2006, where the execution time is normalized to Simics running without caches enabled in JIT mode.
Chapter 5

Branch Predictor Implementation

The following part of this thesis work presents the implementation of a simple branch predictor. Simics currently does not model any branch predictor. In Section 5.1 the model implementation is introduced. No branch predictor extension is currently shipped with Simics. At first, the implementation with the new instrumentation API is presented in Section 5.2. In Section 5.3 the model is optimized with the techniques already presented for the cache model in 4.5. Eventually, the performance of the model is analyzed in Section 5.4.

5.1 Model Specification

The implemented branch predictor has a simplistic structure to keep the overhead significantly low. The algorithm chosen for the model is gshare (5.1), it is a well-known algorithm that collects all the relevant data to perform branch predictor simulation. The objective is to develop a fast simulator whose speedup techniques can be applied to different existing algorithms[10].

The model records branches history into a shift register and the address of the current branch and the branch register are used to perform a XOR and index a table of prediction bits. The table is directly indexed and the two prediction bits are used to predict the outcome of the branch [4][10].

5.2 Branch Predictor implemented with the Instrumentation API

Design

As the previously implemented cache model described in Section 4.3, this model uses the new instrumentation API available in Simics. Every conditional branch instruction is tracked. The branch predictor model subscribes for callbacks when one of the following instructions is processed: \{ja, jae, jb, jbe, jcxz, jecxz, jrcxz, je, jg, jge, jl, jle, jne, jno, jns, jo, jp, jpo, js\}. The model uses
the \texttt{cached\_instruction} callback to keep track of the previously mentioned conditional branch instructions whenever they are inserted in Simics internal instruction cache for performance reasons (2.1); this allows the model to register callbacks for individual instructions, i.e., instructions at a specific address. When this callback is installed, the model will be called every time an instruction is put into the internal instruction cache. Afterward the \texttt{register\_instruction\_after} callback is used to be called each time the instruction is executed. The \texttt{instruction\_after} callback is used since it makes possible to access the new PC value and evaluate if the jump instruction was a \textit{fall-through} or \textit{taken}.

**Callback Overhead Analysis**

To access the code that models the branch predictor for every conditional branch is expensive, therefore, further studies address the extent of the overhead caused by the callback.

As highlighted by previous studies for the cache model, it is expensive to go out from JIT code and reach the C code for calling the model. Thus the idea is to implement a buffer that stores the current branch address. Since the full address is not used...
5.3. Buffer and Event Implementation

Figure 5.3: Branch predictor overhead with callbacks for every branch instruction. The execution time is normalized to Simics running without any extension model.

to index the prediction table, the address is shifted so that the least significant bit is set to one or zero depending on the outcome of the branch. The outcome of the branch is already known from the primitive in JIT code, therefore, it is possible to store a taken/not-taken flag in the address being read by the model.

Conditional Branches in JIT

Branch instructions when compiled in JIT follow different paths to reach the model depending on their outcome. The Instrumentation_After primitive handles branches that result as a fall-through, see Figure 5.4. If, on the other hand, a branch results in a jump the primitive Instrumentation_Branch_Relative_After is called instead.

Figure 5.4: Instrumentation_After primitive in JIT code, without optimization.

5.3 Buffer and Event Implementation

The idea is to collect all the conditional branch instructions from JIT into a buffer from which the addresses are read out after N number of steps or after the simulation goes through the interpreter path and a callback is issued. In the JIT code, there are two different primitives for handling jump instructions depending on their outcome, therefore once the functions are accessed, it is possible to encode the outcome in the address. The address is shifted up and the least significant bit is set to zero or one depending on the outcome: a fall-through or a jump.
Chapter 5. Branch Predictor Implementation

Figure 5.5: Instrumentation_After primitive in JIT code with optimization.

In 5.5 is shown how it is possible to get hold of the physical address of the branch from its virtual address using the page offset. The address is stored in the buffer and the pointer is moved to the next position. The buffer is implemented in the cache model but it is accessible through an ad-hoc interface for communication between the cache model and the processor model. Firstly, when the buffer is read out, the least significant bit is extracted to check the outcome of the branch instruction and the function shown in Figure 5.6 is called: the current address read from the buffer is XORed with the global history bits to index a table of prediction bits. The predicted outcome is checked against the real outcome to update the statistics. Next, both the prediction bits of the table and the global history, are updated with the current branch outcome.

```
static void
make_prediction(branch_pred_t *bp, uint64 pc, uint8 outcome)
{
    uint64 index;
    uint64 pc_bits;
    uint64 hist_bits;
    uint64 prediction;

    pc_bits = pc & bp->conf.mask;
    hist_bits = bp->ghistory & bp->conf.mask;
    index = hist_bits ^ pc_bits;
    prediction = bp->history[index];

    /* Update statistics */
    bp->stat.tot_branch++;
    if (outcome == 1 && prediction == 1)
        bp->stat.branch_misp++;
    else if (outcome == 0 && prediction == 1)
        bp->stat.branch_misp++;

    /* Train predictor */
    if (outcome == 1 && bp->ghistory[index] < 3) {
        bp->ghistory[index]++;
    } else if (outcome == 0 && bp->ghistory[index] > 0) {
        bp->ghistory[index]--;
    }
    bp->ghistory = (bp->ghistory <= 1) | outcome;
}
```

Figure 5.6: Branch predictor implementation make_prediction function.

5.4 Performance Results

To evaluate performance gain, the slowdown of benchmarks in SPEC 2016 CPU are compared and the execution time is normalized to Simics running without any extension. In Figure 5.7 the slowdown is given by the model reading out the buffer...
after 10 thousand steps. The buffer is emptied in case the model receives a callback from the interpreter path where the optimization has not been implemented.

Figure 5.7: Branch predictor slowdown comparison between the implementation with a callback for each branch address and batched instructions. The execution time is normalized to Simics running without any extension model.

The optimization with 10k batched instructions results in a significant speedup compared to the model implemented with a callback for each conditional branch instruction as reported in Figure 5.7.
Chapter 6

Future Work

The idea is to run those models in a separate thread to further explore the speedup of the newly developed models. The expected outcome of this implementation is to achieve a slowdown whose only constraint is defined by the overhead of collecting data in the buffers. However, another possible outcome could be that the slowdown due to the models is the bottleneck. In this case, new optimized methods for computing statistics in the device models are needed. Moreover with a multiprocessor system the line buffer needs to be cleared out before switching processor; Simics CPUs are simulated in a round-robin fashion, each processes a specific number of instructions at a time, therefore the developed cache model would be an approximation due to different interleaving among different processors.
Chapter 7

Conclusion

This thesis introduces a new fast way of collecting the necessary data for cache and branch predictor simulation using Simics virtual platform. Benchmarks from SPEC CPU 2006 benchmark suites were used to measure the slowdown of the simulation.

The evaluation has shown that the new model implemented with the instrumentation API performs orders of magnitude better than the model already existing in Simics. The initial model was analyzed and optimized leading to a solution where the overhead introduced by collecting the necessary information for every data access and instruction fetch is extremely low. From the results of the benchmarks examined, various optimization ideas were in fact approached. The results show that when the models use the new instrumentation API, there is a significant improvement in the performance compared to the model implemented with the timing model interface. Moreover, for the cache model, when using an L0 cache directly implemented in the processor structure, a buffer for batching both instructions and data accesses and an event that triggers the computation of the statistics in the cache model, the slowdown is further reduced.

The same optimization can be applied to the branch predictor model where collecting data for branch prediction simulation introduces minimum overhead and the full simulation is on average less than 2 times slower compared to the simulation with no extension model attached.
Bibliography


