NTNU
Det skapende universitet

Evaluering av Chip Multiprosessor Simulatorer

Arnt Jørgen Lande

Master i datateknikk
Oppgaven levert: Juli 2006
Hovedveileder: Lasse Natvig, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Oppgavetekst

Multiprosessorer realisert på en brikke er en arkitektur med økende popularitet, og mange prosessor-leverandører tilbyr i dag flerkjerne prosessorer. Dette skyldes at utviklingen av mikroprosessorer møter mange store vanskeligheter: (a) høyt effektforbruk og varmeutvikling, (b) ekstrem kompleksitet, og (c) stort ytelsesgap mellom prosessor- og minne-teknologi. Samtidig blir det plass til mer funksjonalitet eller flere like prosessorer inne på samme brikke. Mange tror at chip multiprosessorer vil kunne redusere mange av disse vanskelighetene.

For datamaskingruppas forskning på temaet Chip Multiprosessorer (CMP) er simulering av slike arkitekturer sentralt. Det finnes i dag et mangfold av forskjellige CMP-simulatorer, og oppgaven går ut på å evaluere et utvalg av disse.

Stikkord for hva evalueringen kan inneholde er:
- Utvidbarhet / Er det mulig å endre på arkitekturen?
- Hvilke instruksjonsett (ISA) er støttet?
- Hvilke benchmarks er det mulig å kjøre?
- Hvilke minne- og multiprosessor-arkitekturer er støttet?
- Hvilke typer parallelitet er støttet?
- Cache koherens
- Ytelse
- Tilgjengelighet
- Nøyaktighet

Oppgaven bør inneholde de mest aktuelle av disse simulatorene: RSim, SimpleMP, ASIM, SimOS, Simics, TFSim, og SimFlex. Vår egen simplescalar-baserte CMP simulator skal også evalueres. Det kan også bli aktuelt å utføre forbedringer av denne.

Oppgaven gitt: 20. januar 2006
Hovedveileder: Lasse Natvig, IDI

Abstract

This thesis presents some of the simulators that are available for simulation of computer architectures, with a special emphasis on simulating chip multiprocessor (CMP) architectures. The simulators Rsim, Asim, SimOS, Simics, TFsim, SimFlex, GEMS and M5 are described, in addition to an extension to SimpleScalar written at the department.

The simulators have been evaluated according to various criteria, such as availability, extensibility, simulation platform, etc. The simulator M5, which has been selected as a good choice for use in the group for computer architecture research at the department, has been evaluated in more detail. The simulator has favourable characteristics such as good extensibility through modular design and pervasive object orientation, support for both full system simulation and syscall emulation, an active development team and user forum, and a fair amount of available documentation.

Preface

This master's thesis was written at the Department of Computer and Information Science, Norwegian University of Science and Technology. The problem description is related to the research being done on computer architecture at the department. The work was done from January till July 2006.

I would like to thank my supervisor, Professor Lasse Natvig, for valuable advice during the thesis work. I would also like to thank Marius Grannæs for useful suggestions and technical expertise.

Trondheim, July 14, 2006

Arnt Jørgen Lande

Table of contents

Chapter 1

Introduction

1.1 Multithreaded architectures

Recently, we have seen a number of multicore processors entering the market, some of which I have previously surveyed [Lan05]. The first processors were targeted mainly at servers, where there is a high degree of parallelism inherent in typical applications [SA05, OH05]. Currently, the multicore processors are also conquering the desktop and even notebook markets (just view the model selection in any store selling computer parts).

The popularity of processors with multiple cores is related to major obstacles for continued performance increase for single-threaded processors. Traditionally, Moore's law[1] for processor performance has been true, but the following difficulties make it hard to follow the same path for increased performances in the future:

- There is an almost cubic relationship between CPU frequency and power consumption [SA05], resulting in a high associated cost with increased clock frequency; this is both in terms of power usage, heat dissipation (and associated cooling issues), and possibly battery life.
- The processor-memory gap, which is the difference between the CPU and main memory in performance, is ever-increasing. If the CPU must wait for data from memory, processor cycles are wasted.
- Processor design has grown to a task of extreme complexity. With an enormous amount of transistors on a chip, and the need for dynamic discovery of instruction-level parallelism (instructions that can be executed

[1] Moore's law states that the number of transistors, giving roughly the computing power, will double every 18 months [Wik06].

in parallel without breaking any data dependencies) in program code, it has become more and more expensive to design new processor generations.

- There is also a hard limitation to instruction-level parallelism; the number of instructions that can be executed simultaneously without breaking any data dependencies is usually less than 5 [Wal91].

Chip multithreading (CMT) [SA05] (not to be confused with chip multiprocessor (CMP)) designs have emerged to attack these difficulties. Instead of using the larger number of transistors that a single chip can hold to increase the complexity of the single-threaded architecture, hardware support is added for concurrent execution of multiple threads (processes or programs). There are two main approaches to this:

- *Simultaneous multithreading* (SMT) is the technique where a single CPU is able to maintain multiple thread contexts at the same time [URŠ03], allowing instructions from different threads to be in the execution pipeline simultaneously.
- *Chip multiprocessor* (CMP) design achieves parallel execution of threads by having multiple – but simpler – cores on the same chip instead.

The two techniques are indeed not mutually exclusive; for example, the Sun UltraSPARC T1 consists of multiple processing cores, each implementing SMT [KAO05]. An advantage of the CMP alternative, however, is that older and more or less unmodified cores can be scaled down to the current production technology, and duplicated on a single chip. This leads to lowered development costs. The first multicore processor for desktop computers, the AMD Athlon 64 X2, used a relatively simple CMP design with two processing cores [QTY05].

1.2 Motivation for thesis

CMP architecture is a major research focus of the research group for computer architecture at the Department of Computer and Information Science at NTNU. It is therefore important to have a simulator tool to be able to explore how design choices will affect overall performance. Since it has not been settled on a particular simulator tool yet, an overview and an evaluation of the available choices is desirable. Hence, the purpose of this thesis is to provide that.

Chapter 2

Simulators

2.1 Simulator evaluation criteria

Refer to the problem description for a list of evaluation criteria for the simulators. Some points are easy to determine for any simulator, like for example the ISAs supported and availability. Others are harder to determine, like accuracy, performance and extensibility. Such characteristics can only fully be investigated through thorough testing for the simulator. As I have not been able to do this for all the simulators, I have had to rely on the information that was available about each simulator (of which the amount varied). I will however select one simulator for a closer look and testing (see the next chapter).

2.2 Rsim

Rsim is a simulator for shared-memory multiprocessors, modelling detailed ILP processors. The background for developing Rsim was that prior shared-memory multiprocessor simulators had used too simplistic processor models, while new research showed that shared-memory multiprocessor performance could be improved with the use of techniques such as aggressive implementations of sequential consistency, based on the ILP-exploitation in modern processors [HPRA02].

Rsim was developed at the Rice University, Houston[1] (hence the R), with memory and network subsystems based on the former Rice Parallel Processing Testbed (RPPT). The first release was made public in 1997, free of charge for noncommercial use. The simulator can be downloaded from the Rsim homepage at `http://rsim.cs.uiuc.edu/rsim`.

[1] The Rsim group is no longer based at Rice, the group homepage at `http://rsim.cs.uiuc.edu/rsim/` lists people from several places as members.

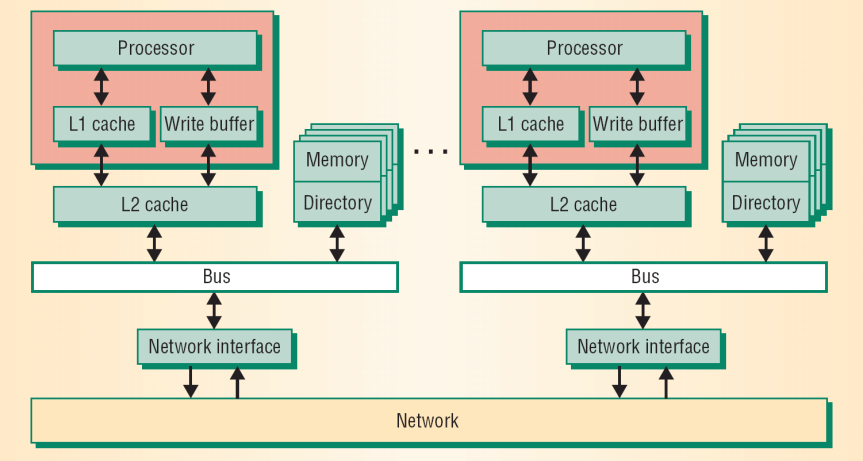

Figure 2.1: Rsim multiprocessor configuration. Figure copied from Hughes *et al.* [HPRA02].

2.2.1 Simulator system

Rsim is made up of interchangeable modules – thus the range of supported architectures is only limited by the availability of, or willingness to write, new modules for the desired architecture.

Statistics output from the simulator includes total instructions executed, divided into different categories, instructions per cycle achieved, functional unit usage, detailed cache statistics, and more.

Rsim is an execution-drive simulator, and interprets application executables. The instruction format is in the Sparc V9 format, but is expanded internally to a simulator-specific format. Internally the simulator is based on the RPPT Yacsim (Yet another C simulator) library.

2.2.2 Multiprocessor configuration

Figure 2.1 shows the multiprocessor configuration for Rsim. As can be seen, it is a multiprocessor with non-uniform (distributed over each node) shared memory. The cache coherency is directory based, using the coherency protocol MESI or MSI. The interconnect network is a two-dimensional, wormhole-routed mesh network, but this should be configurable.

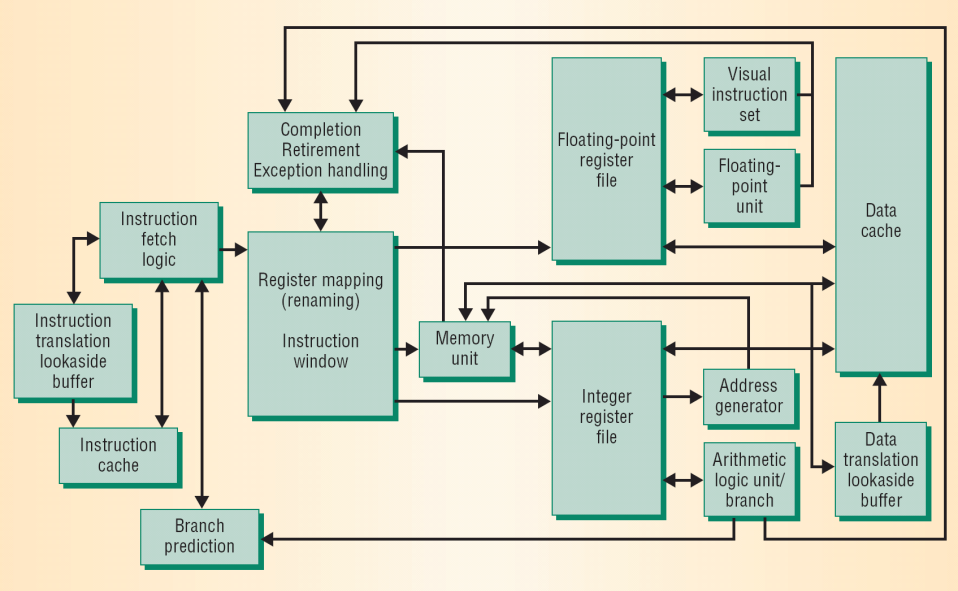

Figure 2.2: Rsim processor architecture. Figure copied from Hughes *et al.* [HPRA02].

2.2.3 Processor architecture

At the time of development, no detailed ILP processor simulator was available, so the development team created a processor model mostly based on a preprint of the MIPS R10000 architecture manual [HPRA02].

The processor architecture is shown in Figure 2.2. The complexity of the processor can be configured – ranging from single-instruction issue, in-order completion, to multiple-issue, out-order completion. It supports register renaming, static and dynamic branch prediction, multimedia instruction set extensions and simultaneous multithreading. Parameters like issue width and number of functional units are user configurable.

2.2.4 Preliminary conclusion

The main problem with Rsim is that it models a shared-memory multiprocessor over network, and not a chip multiprocessor. The usefulness of Rsim depends on how easy it is to replace the network with a bus for inter-chip communication between CPUs.

2.3 SimpleMP

SimpleMP is based on the SimpleScalar simulator, an execution-driven, detailed architectural simulator for uni-processors [ALE02], and extended to support multi-processors [HS00]. Unfortunately, I have not been able to find more information about this simulator.

2.4 Asim

Work with Asim was started at Compaq in 1998. At that time the processor models used for prediction of performance were becoming extremely complex, and there was no structured method of developing these [EAB+02]. Asim is a performance model framework, based on modularity and reusability, that was developed as an answer to this problem. Through modularity, the Asim framework allows a complex hardware unit and its functioning to be broken down into smaller, manageable pieces. Module reusability saves work as it is easy to re-use an already written module in a new project, and this in turn also increases confidence in the given module with its use in different settings.

A *module* in Asim represents a hardware component, such as a cache, or the functioning of a hardware algorithm, such as a cache replacement policy. Modules communicate through method calls or ports, the latter being an abstraction to allow a module to specify its communication and timing characteristics (more about this later). When a user configures a set of modules for running a simulation, these modules are instantiated in a hierarchical system by a configurator (a normal C++ compiler and a makefile). The hierarchical collection of instantiated modules, together with an Asim controller, make up a *performance model*. Thus, a performance model in this respect includes the hardware model that is being simulated, and the characteristics that are being measured (e.g. IPC for a processor, or miss rate for a cache memory). Figure 2.3 illustrates this process.

An important aspect to note about Asim is that these modules are replaceable, as mentioned. This means that Asim is not a simulator for one specific performance model, but rather a collection of tools for creating and running different performance models. To help with these processes, the Asim framework comes with a set of tools called the "Architect's Workbench" (AWB). The basis for this workbench are AWB configuration files. An AWB file is a complete specification for an experiment, including the performance model, program or benchmark details, and default parameters. The format is a structured text file.

The other main part of the Architect's Workbench is a graphical user interface for manipulating the AWB files. The GUI lets the user select modules from a pool of available modules, starting with a top Asim system node, and proceeding

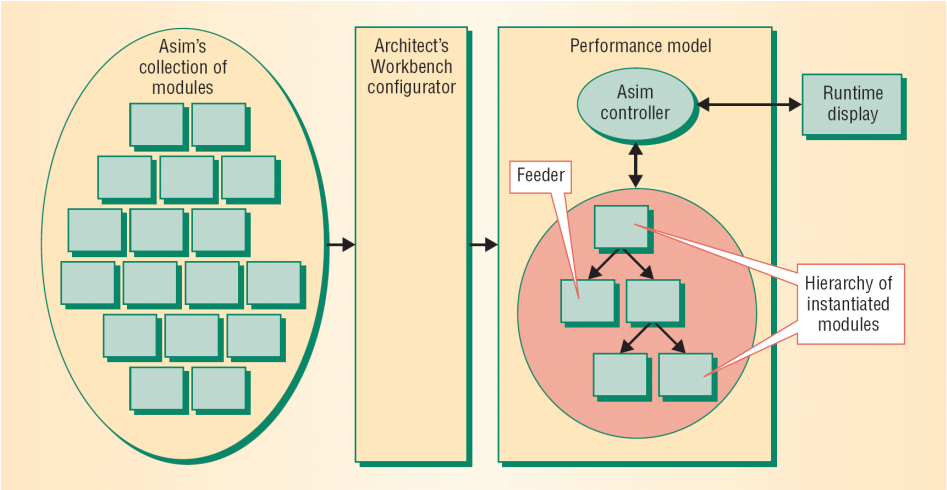

Figure 2.3: Asim framework. A user configures a set of modules to make up the simulation by using the Architect's Workbench, which are in turn instantiated in a hierarchy and run by a controller. During simulation, the runtime display shows the actions within the modules on a cycle-by-cycle basis. Figure copied from Emer *et al.* [EAB+02].

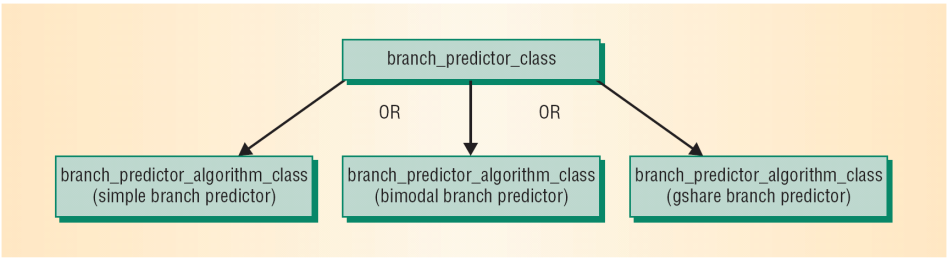

Figure 2.4: An example of different implementations of a branch predictor module. Figure copied from Emer *et al.* [EAB+02].

downwards in a hierarchical fashion. The AWB can then generate a build tree for the model, which can later be run as an experiment.

2.4.1 Modules

The Asim modules that represent hardware structures or functions, are written in the C++ language. For design alternatives of particular functions, Asim does not rely on the `#ifdef ... #endif` feature, which lacks clearly specified code boundaries, but lets instead each module have its own C++ class.

A module/class representing a hardware unit/function typically inherits a standard `asim_module_class`. The module can then be broken down into submod-

ules, with the respective submodules representing different implementations of that particular module. This concept is illustrated in Figure 2.4. In this case, all the submodules must conform to an agreed upon interface – namely, particular method calls. This is called a *method-call interface* style.

The other interface style is called *ports*. This is communication between hardware units represented as modules, and is realized by having each module implementing a clock method that handles sending and receiving of information in one cycle. The sender/recipient of the information is decided from an identifier string. Asim utility code then handles the actual connection of modules through ports, and exchange of information, based on the identifier strings. Ports have the important properties of fixed delays and fixed bandwidth. Because of this, Asim models can take into account the wire delays in physical circuits.

A *feeder* is a special kind of module that does not represent a hardware unit. Feeders are responsible for supplying input to the performance model – that is, instructions to execute. Asim supports three types of feeders:

- The *static trace feeder* reads instruction traces that could either have been generated by another performance model, or a real machine, and feeds them to the performance model. (In other words, the simulation is trace-driven.)
- The *dynamic trace feeder* emulates instructions to generate a trace on-the-fly. For this purpose, it uses the SimOS full-system simulator to read instructions from.
- Finally, the *Aint feeder* supplies the instructions from a program binary, compiled for the target (simulated) architecture. More specific, the performance model instructs the Aint feeder to fetch and execute instructions, the feeder also maintaining the architectural states. The Aint feeder is slower than its alternatives, but can, together with the performance model, simulate a modern, speculative processor. (This option represents execution-driven simulation.)

2.4.2 Preliminary conclusion

Asim, with the appropriate modules available, is an interesting simulator. For a multiprocessor, it would be sufficient to simply instantiate multiple copies of the processor. However, Asim does not seem to be available for download and free usage. The simulator is a proprietary tool within Compaq and Intel [MAA+02], and would therefore be hard to obtain.

2.5 SimOS

The developers of SimOS saw the need for a simulator that is able to simulate not only user applications, but also the operating system [RHWG95]. The reason for this is that applications depending heavily on operating system services will not be well understood when the simulation focuses only on the user application. Second, operating system designers would like to be able to test their own systems on simulators, in the case of unavailable or even non-existing hardware.

The SimOS simulator aims to solve this problem by being able to simulate an entire operating system with user applications. For the simulation time not to be overwhelmingly long, SimOS relies on the following two advantages:

- SimOS provides extremely fast simulation of hardware: a factor of only 10 times or less slower than native hardware execution should be possible.
- It is possible to control the level of detail for the hardware simulation. That means that the researcher can "fast forward" the simulation, using SimOS fastest simulator (or direct execution), to a place of interest, and from there continue simulation at a slower pace and in more detail.

SimOS development was initiated in 1992, and was taken into usage in 1994, at the Stanford University. The simulator has a homepage at `http://simos.stanford.edu`. It is last updated in 2005, which shows that there might still be some activity regarding the simulator. The latest release, however, version 2.0, is from 1998. New in the second release was support for, among others, DEC Alpha/Unix target system, and a port of the simulator to the x86/Linux platform. SimOS can be downloaded from the homepage by filling out a registration form.

2.5.1 Direct execution

The direct execution mode is the fastest mode of execution within SimOS. When the target system (simulated system) and host system (system upon which SimOS runs) are similar, it is possible to exploit the similarity by executing simulated instructions at near native speed directly on the hardware. This is not useful for experiments, because most statistics and details from the execution will be lost (one might as well just run the application on native hardware). It can, however, be a useful means for fast forwarding the simulation to a point of interest (for example run the boot process of an OS).

The direct execution mode was challenging to implement, more about the details of this later.

2.5.2 Detailed simulation

The detailed component simulation (CPU, memory) is provided as a hierarchy of models that support varying levels of detail. For CPU simulation, it ranges from binary translation (on-the-fly translation of application code to host machine code), to detailed software interpretation of instructions.

2.5.3 Preliminary conclusion

SimOS is, however an important simulator when first developed, starting to get old. Since the last release came out in 1998 – eight years ago – a lot has happened, and I believe the similar simulator Simics is a better choice for full system simulation at this time. SimOS still has a relevance, however, as it is used by Asim (see Section 2.4).

2.6 Simics

Simics is a full system simulator like SimOS, but is – in contrast to SimOS – offered commercially through a Swedish company called Virtutech. Simics originates back to a simulator called g88 developed at IBM [MCE+02]. Then work on a simulator based on g88 called gsim, for simulation of shared-memory multiprocessors, started in 1991. Finally, in 1994 gsim was renamed Simics. Simics was first developed as an academic project, but was later transferred to the Virtutech company.

Simics aims, like SimOS, to offer relevance through the ability to run complete and realistic workloads. Magnusson *et al.* [MCE+02] argue that by running small, "toy" workloads with great accuracy, there is a danger of obtaining accurate results to irrelevant questions. Simics therefore attempts to find a balance between accuracy and performance, such that relevant workloads can be run within acceptable amounts of time.

One of the unique features of Simics, is that it is able to simulate hosts of different types (hardware and software) in a heterogeneous network, within the same framework. Figure 2.5 gives an example of this. Distributed over four host computers, Simics simulates four different target architectures and operating systems with software, with good enough performance to use the system interactively. All supported target architectures are simulated with sufficient accuracy to allow device drivers and firmware to run unmodified, and the operating system to install normally.

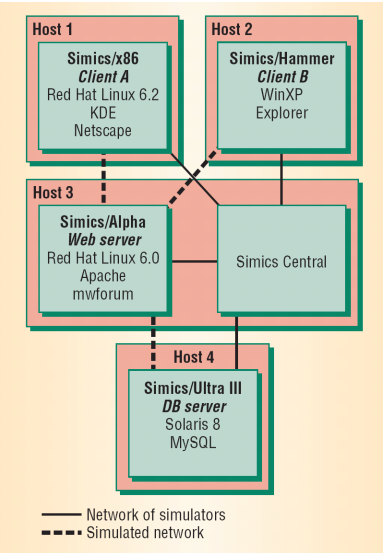

Figure 2.5: An example of a distributed Simics simulation of four target machines of different types. The Simics Central administrates the local clocks and data exchange over a simulated Ethernet link. Figure copied from Magnusson *et al.* [MCE+02].

Simics simulates the following instruction set architectures[2]: UltraSparc, Alpha, x86, x86-64, PowerPC, Itanium, MIPS and ARM, and runs on Linux (x86, PowerPC, Alpha), Solaris/UltraSparc, Tru64/Alpha, and Windows 2000/x86.

2.6.1 Implementation

Simics is a large and complex simulator, with almost a million lines of code, having taken more than 50 person-years to develop. Figure 2.6 shows a picture of the overall system architecture. The core module to the left takes care of the basic simulation tasks, like event handling, memory management and CPU instruction interpretation. Other devices, like disks, network and graphics controllers are connected through the Simics application programmer's interface (API). The same are simulator tools like debugging, scripting and the command line interface that is the main control centre for Simics. Outside is the already mentioned Simics Central that manages clocks and simulated network links when running a distributed simulation and/or simulated distributed system.

The target system (to be simulated) is described in a specific object-oriented language, where an object represents a processor or device, or a virtual object like a disk image. By the use of the Simics API, the developer can describe a

[2] According to the article [MCE+02] from 2002; support for more architectures might have been included at a later time.

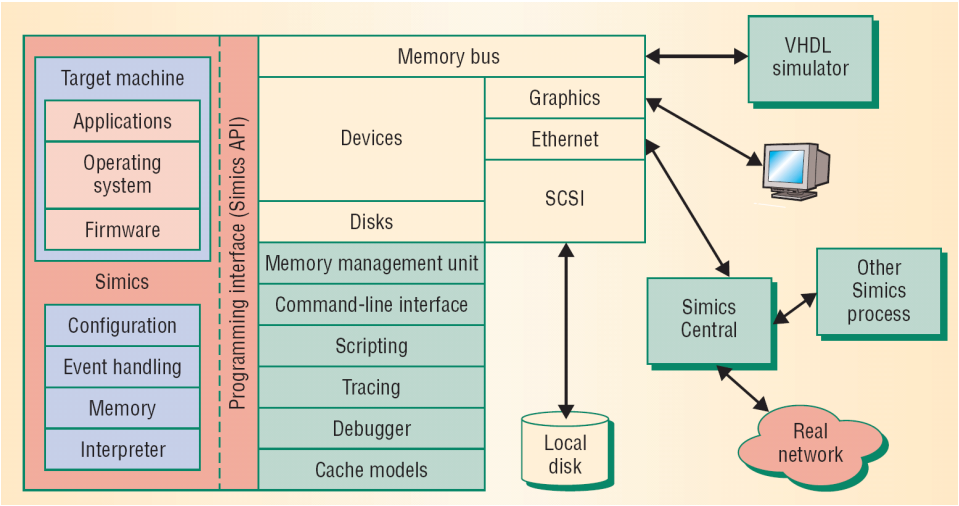

Figure 2.6: Simics architecture. To the left (enclosed by a shaded, gray box) is the core module, which offers basic simulation features to the target system. More modules and tools are available through the Simics API. Especially noteworthy is the Simics Central, mentioned before, and the VHDL simulator that allows any component written in VHDL language to interface with the system bus. Figure copied from Magnusson *et al.* [MCE+02].

module in this language, which are then loaded by Simics. The API is written in C.

The target CPU instruction set architecture (ISA) is coded in a special language, called SimGen. It is a high-level language allowing the developer to describe the format, syntax, semantics (functionality) and attributes used with timing models for the ISA.

2.6.2 Preliminary conclusion

Simics' main strength is its versatility in support of many different target and host architectures, and ability to run complete and real workloads, including entire operating systems and user applications. Simics is by itself not able to simulate advanced, out-of-order processing cores with enough detail to experiment with CMP microarchitecture, which is done by the research group at IDI.

Another important advantage of Simics is that it is commercially available, which ensures it is updated and maintained regularly. And IDI has a license for using it.

2.7 TFsim

TFsim – timing-first simulator – was developed part-time over one and a half year, by a graduating student at the University of Wisconsin, Madison [MHW02]. "Timing-first can be viewed as an almost correct integrated simulator followed by a correct functional simulator checker" [MHW02]. It is a full system simulator that decouples timing from functional simulation. By simulating timing first, and then verifying against a functional simulator, a number of advantages are gained according to the paper by Mauer, Hill and Wood [MHW02]:

- The timing simulator can ignore certain factors that do not contribute to the total execution time to a large degree – for example PCI bus transfers.
- Development time is reduced for the simulator itself, as existing simulators can be used.
- Timing-first simulators can recover from *deviations* (from the functionally correct simulation), and supply approximations of timing in situations where normal simulators would fail.

It is Simics that supplies the functional simulation to TFsim. TFsim combines Simics with a portable timing simulator written in C++, that compiles on the x86 and SPARC platforms. The timing simulator implements a reduced version of the SPARC V9 instruction set, and does not implement full system functionality including external devices, like Simics does (the idea of timing-first is rather to approximate the correct performance). Through Simics' API, the timing simulator can advance Simics a specific number of steps/cycles, and read its architectural state for verification against the timing simulation's internal state.

2.7.1 Preliminary conclusion

This simulator is based on a novel concept, but as a student project it may not be as updated and error-free as one wishes. Also, the concept in itself is a bit unusual, and it is not clear that this is the right approach for simulation of CMPs at IDI.

2.8 SimFlex

The motivation behind SimFlex is that with research on server systems shifting from scientific to commercial workloads, and processors are increasingly complex, implementing full-system simulators has grown to "a task of herculean proportions" [HSW+04]. The Simics and SimOS simulators are referred to as examples of this. A result is that researchers must base conclusions from these simulations on only a fraction of a second's native execution time, because of

the slowness of these complex simulators compared to the real hardware.

The relation with Simics is important, as SimFlex uses system emulation features from that simulator as a basis for SimFlex' own simulation. In this way, SimFlex can be seen as an extension to Simics. SimFlex is a full-system simulator just as Simics, but implements two innovations that are meant to assess the problems mentioned in the previous paragraph:

- **Compile-time component interconnection** uses standard programming features of C++ to eliminate the software overhead associated with modular simulator design. This is achieved by expressing component interconnections at compile-time, and was inspired by the Asim simulator.
- **Simulation sampling** uses developed SMARTS methodology to select samples for representative measures of a workload, rather than simulating all of it.

The "full-system" terms implies that SimFlex can run realistic, commercial workloads – such as booting a contemporary operating system – and that it simulates also operating system code and peripheral devices, factors that are significant for the overall system performance.

SimFlex is offered for download on the simulator webpage, `http://www.ece.cmu.edu/~simflex`, through the Flexus software. Flexus is a family of component-based C++ computer architecture simulators, built on top of Simics, making up a framework for full-system simulation [WW05]. The first release of Flexus, version 1.0, came out summer 2005.

2.8.1 Component-based design

The SimFlex simulator is made up of *components* that normally correspond directly to a hardware unit, such as a cache. These are connected with something called *wiring*, which is interconnection expressed in C++ language syntax. In this way, wiring descriptions are taken into account at compile-time, and a custom simulation binary for that particular wiring is produced.

The C++ feature in use is the generic concept of templates. Components are then templates with specification of ports to other components, making up the communication paths. In this way, function calls happen directly between components at run-time, and the need for using a global table for wire lookup is saved.

2.8.2 Measurement with Smarts

A simulator such as SimpleScalar is 4000 times slower than the hardware it models, and a multiprocessor simulator is even slower. For that reason, researchers often skip a large part of the instructions of benchmarks when drawing conclusions, simply because simulating the entire benchmark takes a prohibitive amount of time.

Smarts stands for Sampling Microarchitecture Simulation. In short, it is a method of obtaining representative results from executing a benchmark, without having to execute all of it. The principle is to select a minimal subset of the benchmark, such that the results lie within a given confidence interval. The basis for the selection is the statistical concept of variation. The subsets can very well be distributed over the entire instruction span, therefore a way of jumping ahead in the instruction stream is needed. This is achieved by *function warming*, which "warms" the components to the desired microarchitectural state when going execute instructions from a particular location.

2.8.3 Preliminary conclusion

Conclusion will be the same as for Simics with respect to the issue of whether to simulate a full system, and the question of how detailed the processor models are. With reduced execution time and good correctness, this would be an attractive alternative to Simics.

2.9 GEMS

The Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) toolset [MSB+05] builds on two simulators already described here: Simics and TFsim. Like many other simulators, it makes use of the full-system simulation capabilities of Simics, but uses the timing-first approach of TFsim to model a detailed, SPARC V9 processor. Also, particular emphasis is made on the modelling of a cache hierarchy and the interconnection network between nodes in a multiprocessor, experiments with this being a major research objective when creating the simulator toolset. For the purpose of being able to test different cache coherency protocols, they developed the Specification Language for Implementing Cache Coherence (SLICC).

Figure 2.7 shows the GEMS architecture. As can be seen, the heart of the simulator is the Ruby memory system simulator. Ruby can model a hierarchy of caches specific to a single processor, as well as shared caches in a chip multiprocessor. The interconnection network is specified by setting up links

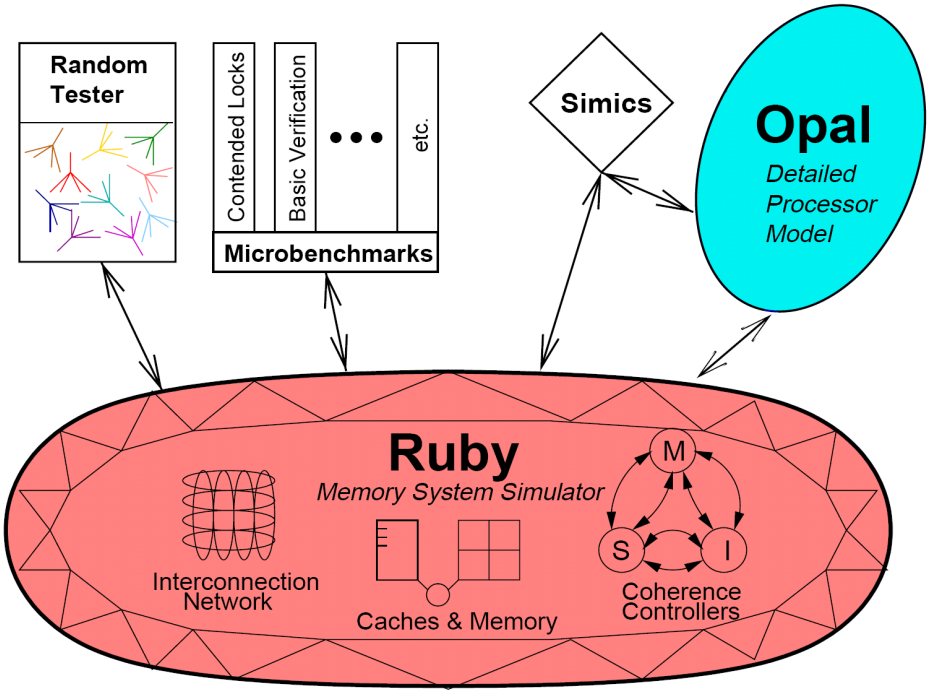

Figure 2.7: GEMS architecture. The heart of the simulator is the Ruby memory system simulator, which can be driven by input from a Random Tester, a set of microbenchmarks, Simics' simple in-order processor or Opal dynamically scheduled processor. Figure copied from Martin *et al.* [MSB+05].

between switches in the source code. The routing tables are then re-calculated for every execution, which eases new network topologies to be added. By default, a switched point-to-point network is modelled in Ruby.

As input to Ruby, one can choose between four modules: a random tester module, which supplies Ruby with random requests with the purpose of stress-testing the memory system; a microbenchmark module which supplies data from microbenchmarks; Simics and its simple in-order processor model; and finally Opal (together with Simics) which is formerly TFsim and is a timing model of a detailed out-of-order processor.

Currently GEMS is released in version 1.2, and is licensed under the GNU GPL license. It is available for download from the simulator's webpage at `http://www.cs.wisc.edu/gems`.

2.9.1 Preliminary conclusion

GEMS seems to be a simulator in active development. The basement upon Simics can both be an advantage and a disadvantage. In the end, the level of detail of the Opal processor model is crucial.

2.10 M5

M5 is a simulation system developed mostly from scratch (only a CPU model is based on SimpleScalar's *sim-outorder* model) [BHR03]. It models a detailed out-of-order processor, an event-driven memory system and detailed network I/O, in a full-system simulator. The motivation for making a new simulator, was that the researchers behind it were not satisfied with available simulator tools, in particular the lack of detailed network I/O simulation in these (TFsim is mentioned in this connection). They argue that while network bandwidth has increased by a hundred times from 1995 to 2002 (from 100 Mbit to 10 Gbit) – an improvement surpassing that of Moore's law for microprocessors – network performance should be an overall system-wide design consideration.

A key property of M5 is the object-oriented modular design, where a component is implemented as a (C++) class, with the consequence that components/classes easily can be replaced or instantiated multiple times. Also, use of "private" and "protected" access to object methods ensure a well-defined interface between simulator components (like would be present in a real implementation). A common superclass to all components, `SimObject`, provides functions for configuration, instantiating and others that are used by all simulator components. It is intended by M5's developers that researchers focusing on a particular field of hardware design can create M5 classes, and share with different focusing teams (not willing to develop a detailed model of that particular component(s)).

An M5 simulation is configured by the use of configuration files written in the Python programming language. Python code then generate textual configuration files following the same style as Windows' ".ini" files. The configuration files are divided into sections, where a section represents a node (component like CPU or cache), and the following keyword/value pairs describe that component. Starting from an obligatory `Universe` node, proceeding down an example path through the hierarchy are the nodes `System` (there can be multiple systems/-computers in a single simulation), `CPU` and `L1Cache`.

There are two available processor models in M5, and a third random data reader/writer called *MemTest*. *SimpleCPU* is a simple in-order CPU model similar to SimpleScalar's *sim-safe*, while *FullCPU* is the mentioned detailed, out-of-order

processor also capable of SMT. The model originates from SimpleScalar's *sim-outorder* model, but is in the process of being entirely rewritten. The simulated ISA is Alpha. Two cache models have been implemented: an LRU cache and an indirect index cache.

For evaluation of M5, the benchmarks *Netperf* and *SPECweb99* were used. Netperf is a simple tool that generates traffic for finding the maximum bandwidth of the TCP connection between machines, while SPECweb99 is a more realistic benchmark that generates HTTP requests from a web server.

M5 models a DEC Tsunami system and can boot Linux 2.4/2.6 and FreeBSD[3] [Mic06]. It runs on x86-systems with Linux, OpenBSD or Cygwin, and is available for free download and use through the simulator's page at Sourceforge, and contains about 90,000 lines of C++ code. The simulator web page can be found at `http://m5.eecs.umich.edu`.

2.10.1 Preliminary conclusion

M5 seems like a very interesting simulator. Since the simulator paper [BHR03] was written, a lot has happened with this simulator. See the last section of this chapter for more discussion on M5.

2.11 IDI simulator

Haakon Dybdahl and Marius Grannæs, at the Department of Computer and Information Science at NTNU in Trondheim, have extended SimpleScalar's *sim-outorder* model to support CMP simulation. I have previously described this simulator [Lan05], but since then some fixes have been applied, and I will give some updated information here.

Support for multiple CPUs is implemented through locks around the simulated pipeline loop, that ensure inter-CPU synchronization on a clock cycle-basis, and allocated shared memory segments to each simulated CPU simulate physically shared memory. The simulation is cycle-accurate. Figure 2.8 shows the simulated memory hierarchy, which features independent CPUs and L1 cache, and shared L2 cache and main memory.

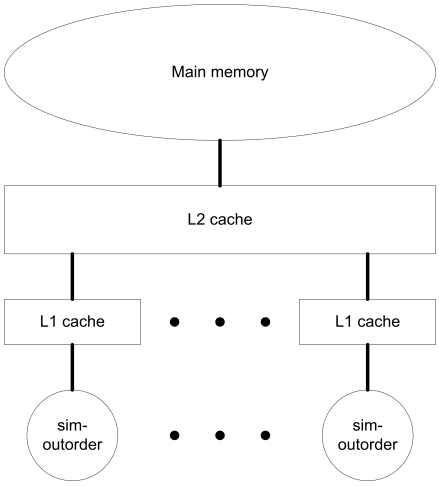

Figure 2.8: CMP simulator memory hierarchy.

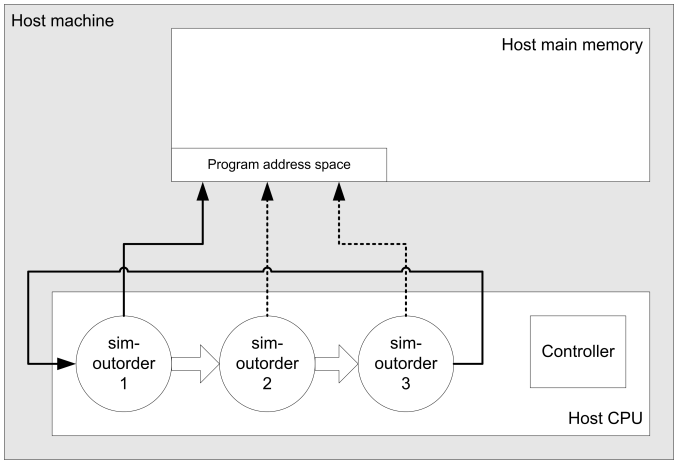

Figure 2.9: CMP simulator implementation. Example with a single-processor host, simulating three CPUs (instances of *sim-outorder*).

2.11.1 Implementation

Figure 2.9 shows how the simulator runs inside the host machine. All simulated CPUs run inside the host CPU, in a round-robin fashion. The horizontal arrows between *sim-outorder* instances symbol control flow between CPUs (implemented by semaphore locks). The arrays pointing from the CPUs to main memory mean that the CPUs communicate with the memory in turns. Finally, the Controller block represents the controller, that can be used to implement some desired management function.

In the method `sim_main()` in `sim-outorder.c` the following relevant steps happen:

1. Synchronization (inter-CPU), controller and report semaphores are set up.
2. Every CPU is attached to a shared memory segment.
3. **Main simulation loop:**
   - *Every 10,000 clock ticks the CPU connects to the controller.* This is not required, but can be a handy way of performing desired tasks in a regular interval; an example is the original experiment where thread migration between CPUs was managed during this communication with the controller.
   - *A lock is obtained for the CPU next in line for executing* (represented by the variable `my_cpuid`).
   - *Normal pipeline functions are executed.*
   - *The lock is released and given to the next CPU to execute* (with wraparound on the total number of CPUs).
4. After a CPU is finished executing, it continues ticking until all the others are done too.

The controller, implemented in `controller.c`, creates the shared memory regions and initializes the various semaphores. It also contains a controller loop that can perform any task needed (as mentioned above). A helper module for creation and usage of shared memory and semaphores throughout the program is implemented in `shared.{h,c}`.

Changes are also made to the cache implementation in `cache.c`. A new function, `cache_create_shared()`, allocates shared memory regions for cache. The original `cache_create()` also calls this function when creating L1 cache, but then with the argument `-1`, that makes the function act as the original function.

[3] FreeBSD support is preliminary according to release 1.1 news.

2.11.2 Shortages

The CMP-extended SimpleScalar simulator had some known bugs and shortages, some of which are now fixed. In a multiprocessor machine with shared memory, the simulator could not take advantage of the multiple CPUs, due to a bug in the synchronization mechanism. This should now be fixed. More a property than a shortage, is the fact that CPU synchronization on every clock cycle introduces significant overhead in execution time. It is probably possible to allow each CPU to execute a few cycles more every turn – to speed up the total simulation time – without too much reduction in correctness.

Another inaccuracy was present in the simulation of L2 cache. Hit rates in the cache will be artificially high because of overlapping address space of CPUs' programs and non-present competition for L2 cache. This was fixed earlier this year, and should not be an issue anymore.

2.11.3 Preliminary conclusion

IDI's simulator is based on the well-known SimpleScalar, which is widely used, but lacks many things, such as documentation, easy interface for extensions and object orientation. To take a step forward, I think it would be wise to look for a new simulator tool.

2.12 Summary and selection

Table 2.1 gives a short summary and comparison of all the simulators described. However, there was a limited amount of information available about each simulator. For a more accurate evaluation of a simulator, a closer look would be necessary, involving study of source code, and practical experiments.

Unfortunately, there was not enough time to go deeply into every one of these 10 simulators, so a selection was required. Ideally, this would be the simulator that was best suited for the research going on at the computer architecture research group at IDI. Considering my impressions from my studies, and a discussion with people from the research group, we landed on M5 as the simulator of choice for my thesis; although there were also good arguments for other simulators, especially GEMS. Reasons for choosing M5 included:

- The processor model in M5 started out from SimpleScalar's out-of-order processor model, which has already been used for simulations within the group. Hence, there might be chances to integrate some existing code into the new simulator, or at least have some basis for customisation of

the new simulator. (A somewhat weaker argument as the old SimpleScalar processor model is currently being phased out from M5.)

- M5 supports everything we need: modelling of detailed, out-of-order processors, CMP architectures, processors with SMT support, interconnection networks and full system simulation.
- It simulates the Alpha architecture, like has been done previously with SimpleScalar within the group.
- The development team has made emphasis on pervasive object orientation, modularity and configurability while developing on the simulator, which results in a well-structured application, and configuration files that work "out-of-the-box" with little changes required.
- It is fairly well documented (at least compared to most other simulators).
- And last, but not least, it has a relatively large user base, an active development team, frequent posts on the mailing lists that are answered by authors of the simulator; and a new release with many new features is soon to be made available.

	Year started & availability	Level of detail	Simulated platforms	Simulated architecture/MP configuration	Extensibility	Emphasis/novel idea
Rsim	1997, freely available	Out-of-order CPU	SPARC V9/Solaris	Shared-memory MP over network	Through modular design	Simulate adv. CPUs in network
SimpleMP	Can't find	Like SS?	Alpha/PISA?	CMP?	?	Extend SS w/ CMP
Asim	1998, within Compaq/Intel	Speculative CPU	Unknown	All	Through modular design	Ease simulator development
SimOS	1994, freely available	Variable	MIPS/SGI IRIX, Alpha/UNIX (*)	All	Uncertain	Simulate entire system
Simics	1994, commercial	In-order CPU	UltraSparc, Alpha, x86, PowerPC, MIPS, ARM, ++ (*)	All	API provided	Simulate "everything", heterogenous networks
TFsim	2002, can't find	Out-of-order CPU	Like Simics? (*)	CMP	Uncertain	New concept: timing-first
SimFlex	2005, freely available	Out-of-order CPU	SPARC V9 (*)	CMP	Modular/OO design	Shorter runtime by statistical sampling of benchmarks
GEMS	2005, freely available	Out-of-order CPU	SPARC V9 (*)	CMP	Modular/OO design	Simulate intercon., extend Simics w/ detailed CPU
M5	2003, freely available	Out-of-order CPU	Alpha, SPARC, MIPS (*)	CMP, partially SMT	Modular/OO design, Python configuration files	Written from scratch, pervasive OO, configurability
IDI's simulator	2005	SS's out-of-order CPU	Alpha, PISA	CMP	Must be "hacked"	Extend SS with CMP

Table 2.1: Simulator summary table. Abbreviations used: **(*)**: full system simulation, **intercon.**: interconnection, **SS**: SimpleScalar

Chapter 3

M5

Remark: During the finishing of this thesis, a new major version of M5 is about to be released (scheduled for the end of July). As both new documentation has become available, focusing on the new release; and older documentation has been updated, there has been an issue which version (1.1 or 2.0) to discuss here. I have tried to include as updated data as possible, but there has not been time to update all information here with respect to the newly available information the last days of the thesis work. The reader should assume that, unless otherwise stated, the given information is either unchanged from version 1.1 to 2.0, or refers to version 1.1.

All test runnings use version 1.1, as 2.0 will not be released until after the deadline.

3.1 Simulator architecture

Figure 3.1 shows an overview of the M5 architecture – or, to some extent, a simplified general computer architecture, with references to where the different components are located in the source code.

3.1.1 CPU

Three CPU models are featured with M5, one simple and two detailed models [BDH+05]. The simple model is suitable for warming caches and getting to know the simulator. It is an in-order, one cycle per instruction CPU that works both in the syscall and full system simulation modes.

The detailed models are one based on SimpleScalar's *sim-outorder* model, still

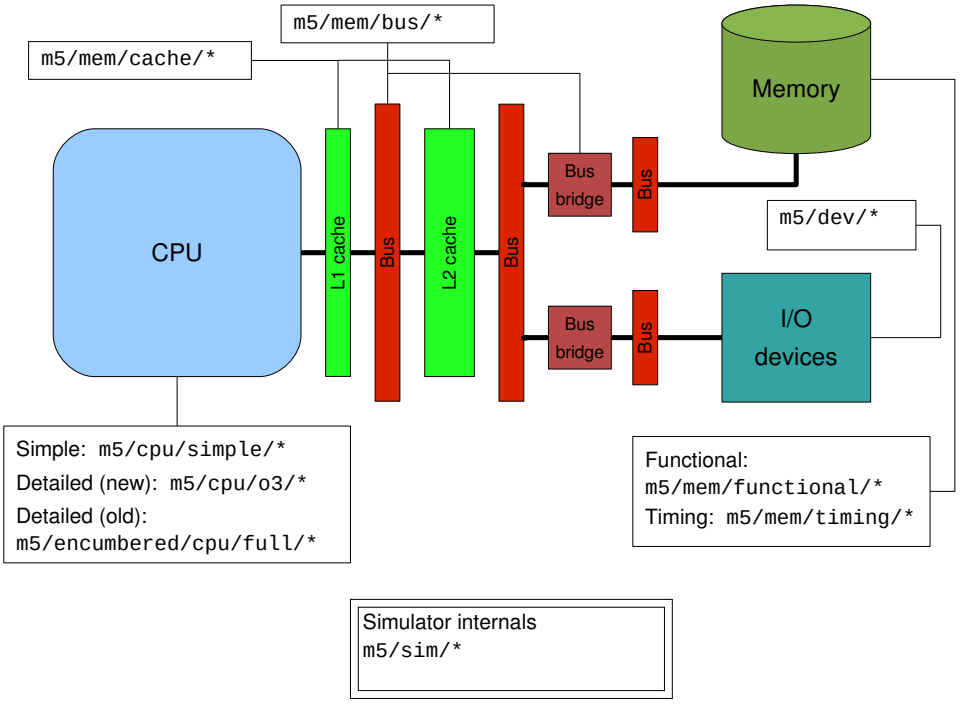

Figure 3.1: An overview of M5 with references to source code. The bold lines are not meant to show a specific form of physical communication, but rather general inter-program communication.

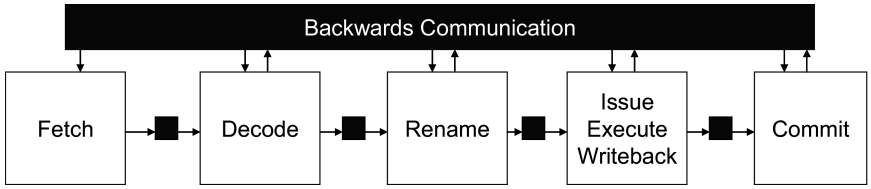

Figure 3.2: Pipeline of detailed CPU model. The forward and backward communication is implemented as time buffers. Figure copied from Binkert *et al.* [BDH+05].

used for full system simulation and SMT support, and a new model, written from scratch. It is the new model that is the emphasis of the M5 team, and support for full system simulation and SMT is planned to be included in this model to make the old model superfluous.

The new detailed CPU model features an out-of-order CPU with a 5-stage pipeline (see Figure 3.2), loosely based on the Alpha 21264. Both forward and backward communication happens through *time buffers*, which is meant to avoid unrealistic interaction between pipeline stages [BDH+05]. The time buffers are similar to queues, and are ticked every cycle. After data is put in the buffer, it can be read out after a specified number of cycles.

A notable difference between the old and the new detailed CPU models is when instruction execution happens. While the old model executes the instruction at fetch, the new model executes it at the execute stage. This allows the timing of each pipeline stage to be modelled.

ISA

Version 1.1 only supports the Alpha ISA, but SPARC and MIPS will be included in the 2.0 release, though only with support in syscall emulation mode. Future plans include full system support for SPARC and MIPS, and possibly new ISAs, like PowerPC and ARM [RBS+06].

M5 includes a custom definition language for ISA descriptions. It gives a compact, human-readable form of ISA descriptions, from which Python classes generate C++ code representing the ISA. Detailed documentation of the C++ instruction objects, the description language and code parsing can be found at the simulator homepage at `http://m5.eecs.umich.edu/wiki/index.php/ISA_description_system`.

The concept of representing hardware instructions as C++ objects means that all information that the simulator needs to execute an instruction, is stored in the instruction object. A parser takes a binary machine instructions, and returns the instruction object. A decoded instruction is stored as a `StaticInst` or a

DynInst class (dynamic for detailed CPU models, holds renamed registers etc.), and is accessed with low overhead later in the simulation.

```
1  decode OPCODE default Unknown::unknown() {
2
3    format LoadAddress {
4      0x08: lda({{ Ra = Rb + disp; }});
5      0x09: ldah({{ Ra = Rb + (disp << 16); }});
6    }
7
8  (...)
9
10   0x11: decode INTFUNC {  // integer logical operations
11
12     0x00: and({{ Rc = Ra & Rb_or_imm; }});
13     0x08: bic({{ Rc = Ra & ~Rb_or_imm; }});
14     0x20: bis({{ Rc = Ra | Rb_or_imm; }});
15     0x28: ornot({{ Rc = Ra | ~Rb_or_imm; }});
16     0x40: xor({{ Rc = Ra ^ Rb_or_imm; }});
17     0x48: eqv({{ Rc = Ra ^ ~Rb_or_imm; }});
18     (...)
```

Listing 3.1: These are samples of the actual decoder specifications. The syntax of the specifications is similar to the C `switch` statement, with a number followed by a colon giving different options (the `switch` keyword is omitted, however). The operations on registers are given in C syntax. `(...)` marks the omission of code.

The ISA description file is divided into two main parts: a part describing the decoder – the component that takes the binary instruction, and returns a C++ object; and a declaration part that describes global information, most notably the instruction formats, and templates for C++ code generation from instruction definitions. Listings 3.1 and 3.2 show example excerpts from both parts of the description file.

The ISA description language compacts the representation to about 1/10: for Alpha 3437 lines of description results in about 39,000 lines of C++ code.

3.1.2 Memory system

A new memory system will be included in the 2.0 release. The most significant change is the melting of the old separate timing and functional models into one unified memory model. Common to both versions is that every memory request happens in terms of a `MemReq` object. After it has been generated by a CPU or another device, the `MemReq` is passed to the right cache on a bus. In case of a cache miss, the cache itself generates a new `MemReq` object that

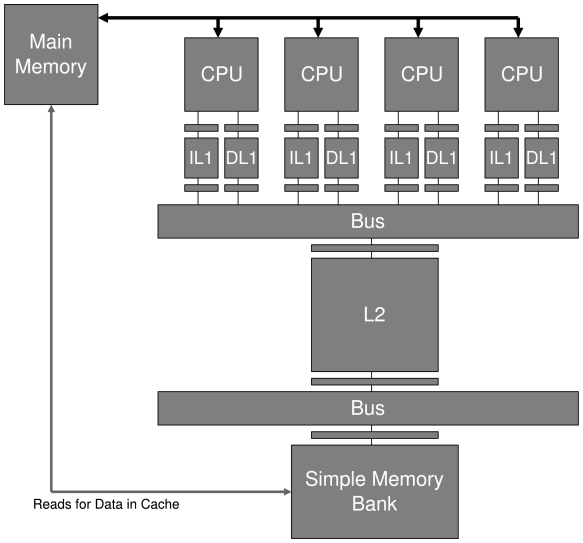

Figure 3.3: M5 memory system in version 1.1. Syscall emulation memory is shown. Figure copied from Binkert *et al.* [BDH+05].

is sent to memory, while holding on to the request until it is able to send a response. Figure 3.3 shows the organisation of the old memory system, while Figure 3.4 shows a new concept called ports in version 2. Ports are a new way of connecting memory devices together, where both parts act as a peer, that can execute send and receive operations on eachother.

The major goals for the new memory system were to

- combine the timing and functional models into one model
- simplify code (e.g. remove duplicate code)
- improve modularity of code (particularly through easier component interconnects)

3.1.3 Cache

Coherency

M5 supports cache coherence in a multiprocessor. The following coherency protocols are supported: MSI, MOSI, MESI, MOESI.

Prefetching

Prefetching is a technique commonly used in computer architecture to reduce the number of cache misses. The concept is to read data or instructions from

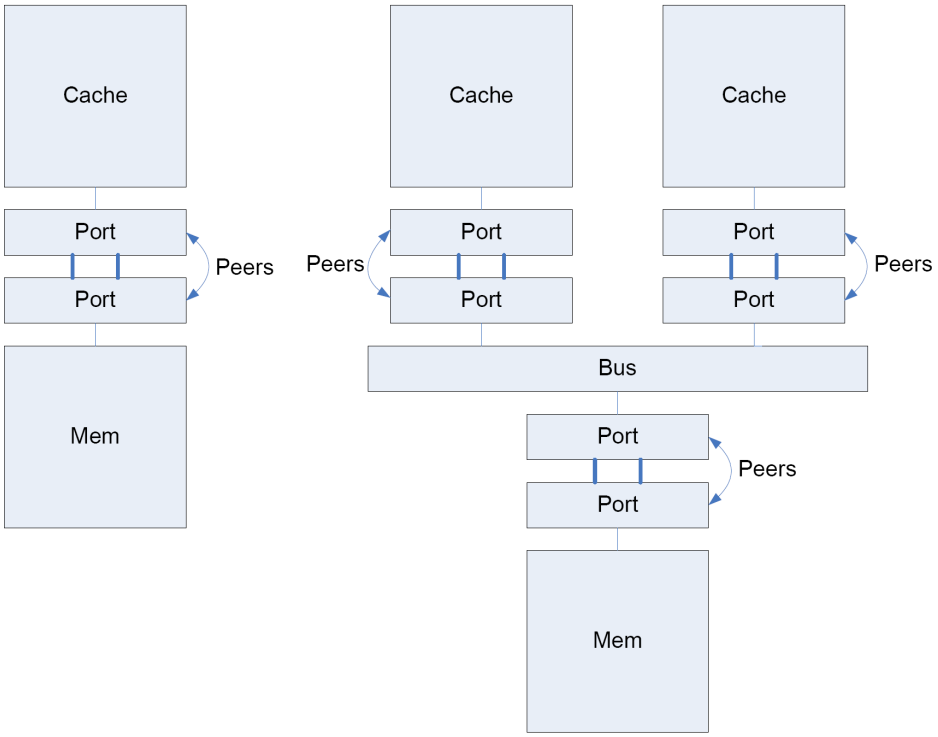
Cache
Cache
Cache
Port
Port
Peers
Port
Port
Peers
Port
Port
Peers
Bus
Mem
Port
Port
Peers
Mem

Figure 3.4: M5.

memory into the cache before they are needed, so that the processor can have fast access to them when needed.

- *Software prefetching* relies on special instructions initiating prefetch, that a programmer or compiler must call at appropriate places in the code.
- With *hardware-based prefetching*, the software is unaware of the data prefetching, and the determination of memory addresses to prefetch is rather based on speculation. The speculation could be as simple as a sequential approach. In that case it is assumed that with the fetch of a particular memory address, it is likely that the consecutive addresses also will be needed by the program. They are then read into the cache.

A modern processor architecture like AMD64 supports both software- and hardware-based prefetching [AMD05].

Tagged prefetching is a hardware prefetching algorithm that associates a bit to each cache block. When a block is fetched, or a prefetched block is referenced for the first time, a prefetch of the next block is requested [WL97]. In an ideal program with only consecutive memory references, this would yield only one cache miss, at the beginning of the program.

M5 supports both software- and hardware-based prefetching. The software prefetching support consists of the Alpha prefetch instructions; though only for data prefetching. With release 1.1 of M5, only one hardware-based policy is provided: a tagged prefetcher. Although references to a stride prefetcher and a GHB (Global History Buffer) prefetcher can be found in the code, they are not yet implemented, and are planned to be included in a later release according to a post by one of the authors to the M5 user mailing list.

M5's tagged prefetcher takes a *degree* parameter that decides how many blocks forward the prefetch goes. There is also an option of whether to prefetch across page boundaries or not, and a latency associated with the operation.

The hardware prefetcher is implemented in `m5/mem/cache/prefetch`, and is based on the template policy for support of various prefetchers. Figure 3.5 shows the implementation hierarchy. There are three levels of inheritance:

1. The base class is `BasePrefetcher`. It provides common functions needed by any hardware prefetcher, but leaves certain functions unimplemented ("virtual" in C++ syntax), to be implemented in specific implementations.
2. `Prefetcher<TagStore, Buffering>` describes a prefetcher with a tag store and a buffer, but leaves the central `calculatePrefetch` function unimplemented.
3. Finally, `TaggedPrefetcher<TagStore, Buffering>` implements the last

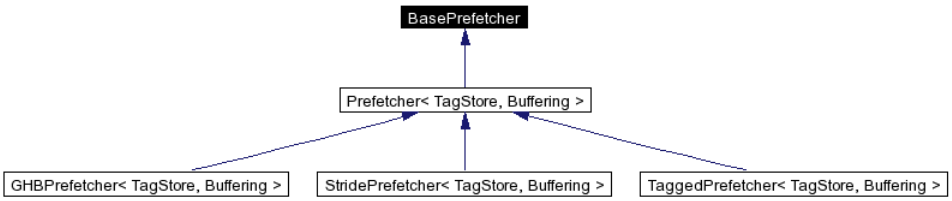

Figure 3.5: M5 prefetcher implementation hierarchy. `GHBPrefetcher` and `StridePrefetcher` are included although they do not come with release 1.1 of M5. This figure is generated by the Doxygen documentation system (described in Section 3.3.1).

function, as a variant of a prefetcher with buffer (other variants are to be implemented).

Listing 3.3 shows the implementation of the prefetch address calculation for the tagged prefetcher. As can be seen, it fetches as many pages as the `degree` parameter says, or until the end of a page is reached, in the case when `pageStop` (a parameter) is true.

To implement a new buffered prefetcher, it would be necessary to create a new class inheriting `Prefetcher<TagStore, Buffering>`, that implements the `calculatePrefetch` function.

3.1.4 Buses

M5 simulates buses between memory, I/O and CPUs in great detail (see sample statistics for bus in the output file in Appendix A). Width and speed can be configured for buses. The interfaces to a bus are called master and slave interfaces, where the master interface sits closer to memory, and slave interface closer to CPU.

3.2 Usage

3.2.1 Building

Compiler

M5 builds with the GNU C++ compiler (g++). The `README` file says that version 3.4 or newer is required; however, there seems to be a problem with the newest version (g++ 4.0.x) related to the usage of templates. If during build one encounters the following error

```
m5/sim/syscall_emul.hh:343: error: there are no arguments
  to open that depend on a template parameter, so a
  declaration of open must be available
```

changing to g++ 3.4 should solve the problem.

SCons

M5 is built using SCons [Kni02]. SCons is a build tool and a replacement for the classic Make, but is intended to solve some of the shortcomings of that tool. These include problems associated with static dependencies, recursive builds of hierarchies and parallel builds in the case of a multiprocessor system. SCons uses Python as the language for the implementation of the build engine, and also the configuration files. Whereas Make was a tool made for programmers, SCons is supposed to be accessible to non-programmers as well, for example scientists. A useful feature of SCons is the simple exploitation of a multiprocessor system during build; by passing the argument "`-j N`" to `scons`, where N is the number of processors available, N jobs are started in parallel.

Build configuration files

As mentioned, the configuration files for the SCons build process are Python scripts. The "top"/main script is called `SConstruct`, and is located in the `m5/build` directory. This script does some elementary checks, like if Python and SCons are installed and of sufficiently recent versions, and determines the build configuration (see next paragraph). It also sets up the build environment, and passes command to the SCons build engine to do its job.

In addition to the top level SCons script, there are scripts called `SConscript` at the following locations:

- `m5/`: This file determines how to build a given configuration from all the source files.
- `m5/python/`: This file is less well documented, but probably deals with including the simulator configuration files (which are also Python files) in the build.
- `m5/libelf/`: The two SCons scripts here configure the headers of the libelf library.
- `m5-test/*test*/`: The scripts in the test directories are not really needed for the build process, but for the regression tests[1] that are included.

[1] Regression tests are intended to ensure that a new version of an application does not break any previously fixed errors. The regression tests provided with the M5 distribution are tested regularly and guaranteed to work by the authors.

Build procedure

M5 builds on Unix-like systems on little-endian hosts, preferably Linux/x86. It can be built in two modes: *system call emulation* (syscall emulation) and *full system simulation*. From `m5/build`, issue the command

```
scons <CONFIG>/<binary>
```

where `<CONFIG>` is either `ALPHA_SE` for syscall emulation or `ALPHA_FS` for full system simulation (both can be built successively to enable both modes), and `<binary>` is either `m5.opt` for the optimised version or `m5.debug` for the debug version.

A slightly different version of the build command is to be used when regression tests should be performed to verify the build[2]:

```
scons ALPHA_SE/test/opt/quick
```

for the syscall mode, and

```
scons ALPHA_FS/test/opt/quick
```

for the full system simulator.

An important note about building the full system simulator: the user must first edit the file `m5-test/SysPaths.py` and update the `SYSTEMDIR` variable to the path to where the full-system binaries are unpacked (these are available for download separately).

3.2.2 Running

To run an M5 simulation, the binary file that was created during build must be supplied with a configuration script, which is a Python file. Such scripts are usually named "`run.py`", and various examples of these come with the distribution. The already mentioned regression tests each has a copy of `run.py`, and these relatively simple tests is a good starting point to get familiar with the format. The syntax of the configuration files is described in the online documentation on `http://m5.eecs.umich.edu/docs/config_files.html`.

It is not entirely true that the Python configuration files are the input to the simulator, because the Python files are translated to Windows-like "`.ini`" files

[2] This seems to be new with version 1.1 of M5, and is described only in the distributed `README` file.

prior to simulation, which the simulator then uses to set up the simulation system. It is planned to remove the necessity for this intermediary step in future releases [BDH+05].

3.3 Documentation

As of present, there are four major sources of documentation for M5:

- The documentation wiki at the simulator homepage (`http://m5.eecs.umich.edu/wiki/index.php/Main_Page`).
- Comments in the source code and Doxygen-generated documentation.
- Paper [BHR03] describing the simulator (although this is not up-to-date with latest versions), and tutorials on M5 (one was held at ISCA-32 [BDH+05], and a new was held at ISCA-33 [RBS+06]).
- The M5 user mailing list, which can be subscribed to and searched from the project page at SourceForge (`http://sourceforge.net/projects/m5sim`). The mailing list is very active, and is a forum for user problems and feedback on the simulator, which frequently receive answers from the simulator authors.

3.3.1 Doxygen

Doxygen is a documentation system for C++ (and more languages) [vH06]. It generates documentation from structured source comments to a number of output formats (both "online" and "offline"), and can also extract relations and generate diagrams from undocumented source code.

Source comments can be in several forms to be recognised by Doxygen. The style used in M5 follows the same pattern as Javadoc comments. For example:

```
/**
 * @file
 * Describes a tagged prefetcher based on template policies.
 */
```

where `@file` is a tag telling Doxygen what is described.

An online version of the Doxygen documentation for M5 can be found at the simulator's webpage. It contains a full list of classes, files, directories and corresponding members. The lists are generated directly from the source codes, and

are thus complete. But the descriptions depend of course on written comments, and are so far highly incomplete.

Apart from listing classes with members, files and more, Doxygen generates class and collaboration diagrams for classes. Inheritance hierarchy and collaboration between a few classes can easily seen from these diagrams; though it does not give an overview of larger parts of the code. There are also some useful short guides of different subjects on the docs page, like how to compile and run M5; the memory hierarchy explained; and guidelines for coding style and documenting the simulator.

New in June 2006 was that these documentation pages, together with all of the simulator webpage, have been converted to the Wiki[3] format. Since the conversion to Wiki, the M5 homepage has frequently been updated, and the online documentation has now been extended, including a new FAQ page and more.

[3]Wiki is a concept where users can easily cooperate in editing web pages, like the Wikipedia encyclopedia.

```
1  // Universal (format-independent) fields
2  def bitfield OPCODE <31:26>;
3
4  (...)
5
6  def operand_types {{
7    'sb' : ('signed int', 8),
8    (...)
9
10 // Basic instruction class constructor template.
11 def template BasicConstructor {{
12   inline %(class_name)s::%(class_name)s(MachInst machInst
       )
13     : %(base_class)s("%(mnemonic)s", machInst, %(op_class
         )s)
14   {
15     %(constructor)s;
16   }
17 }};
18
19 (...)
20
21 // The most basic instruction format... used only for a
     few misc. insts
22 def format BasicOperate(code, *flags) {{
23   iop = InstObjParams(name, Name, 'AlphaStaticInst',
       CodeBlock(code), flags)
24   header_output = BasicDeclare.subst(iop)
25   decoder_output = BasicConstructor.subst(iop)
26   decode_block = BasicDecode.subst(iop)
27   exec_output = BasicExecute.subst(iop)
28 }};
```

Listing 3.2: Samples of the definitions of bitfields, operand types, templates and instruction formats. The code is in Python, except for the output code in the template, which is C++. That is, C++ code is generated by the substitution of certain parameters, contained in the `InstObjParams` object. The substitution is carried out by Python functions. The various `header_output`, `decoder_output` etc. represent different C++ objects to be created.

```
1  void calculatePrefetch(MemReqPtr &req, std::list<Addr> &
       addresses, std::list<Tick> &delays)
2  {
3    Addr blkAddr = req->paddr & ~(Addr)(this->blkSize-1);
4
5    for (int d=1; d <= degree; d++) {
6      Addr newAddr = blkAddr + d*(this->blkSize);
7      if (this->pageStop &&
8        (blkAddr & ~(TheISA::VMPageSize - 1)) !=
9        (newAddr & ~(TheISA::VMPageSize - 1)))
10     {
11       //Spanned the page, so now stop
12       this->pfSpanPage += degree - d + 1;
13       return;
14     }
15     else
16     {
17       addresses.push_back(newAddr);
18       delays.push_back(latency);
19     }
20   }
21 }
```

Listing 3.3: Function in class TaggedPrefetcher that does the actual calculation of memory addresses to prefetch.

Chapter 4

Experiments

4.1 Sample runs of M5

To show practical use of the M5 simulator, I will in this section show some test runs, how they were initiated, what output was produced, etc. As I have not been able to compile any binaries for the Alpha architecture that M5 runs, I rely on the ready-made test binaries that the simulator is shipped with.

There was expressed interest in running the SPECweb benchmark from the computer architecture research group at IDI, but unfortunately the authors of M5 are unable to distribute the SPECweb file image due to legal reasons. It should however be easy to get the benchmark up and running once having acquired the necessary files, as it is used by the development team.

4.1.1 Test 1: four-threaded uniprocessor

The regression tests that come with M5 are guaranteed to work with every release. This is syscall emulation test 4, and can be found in the directory `m5-test/test4/`. The test features a detailed uniprocessor running a four-threaded workload: anagram and gcc, two copies of each.

The test is simply started by typing (from the `m5/build/` directory):

```
ALPHA_SE/m5.opt -d output/ m5-test/test4/run.py
```

("`SE`" is for syscall emulation, and the "`-d`" option makes the output configuration and statistics files be put in the specified directory.)

The `run.py` file is interesting; it defines in a simple way the system, and the

workload. Listing 4.1 shows the file in its entirety. The shortness of the file is possible because it depends on `DetailedUniConfig`, defined in the `m5-test/` directory, which again makes use of ready-made configuration objects that can be found in `m5/python/objects/`. The objects' properties are inherited, as in standard class inheritance, and it is only necessary to specify the details which should be different. In this case, the benchmarks are given, and a maximum instruction count.

```
1 from m5 import *
2 AddToPath('..')
3 from DetailedUniConfig import *
4 import Benchmarks
5
6 BaseCPU.workload = [ Benchmarks.AnagramLongCP(),
     Benchmarks.GCCLongCP(), Benchmarks.AnagramLong(),
     Benchmarks.GCCLong() ]
7 BaseCPU.max_insts_any_thread = 1000000
8 root = DetailedStandAlone()
```

Listing 4.1: `m5-test/test4/run.py`. Configuration file for simulation.

Statistics from the execution is dumped in the file `m5stats.txt`. Listing 4.2 shows a few statistics from this test, while the entire statistics file provides very detailed information; it counts over 5,000 lines.

```
---------- Begin Simulation Statistics ----------

cpu.COM:IPC                                    4.123961
                # Committed instructions per cycle
cpu.COM:count                                   2726458
                # Number of instructions committed
cpu.dcache.read_hits                             754953
                # number of read hits
cpu.l2.read_miss_rate                          0.338584
                # miss rate for read accesses
toMemBus.data_requests                             8920
                # number of transmissions on bus
```

Listing 4.2: A small sample of the statistics output from test 4.

To say something about the performance of the simulator, the statistics file gives some figures for the host computer, namely `host_inst_rate`, `host_mem_usage`, `host_seconds` and `host_tick_rate`. On a Pentium M 1.7 GHz with 768 MB of RAM, it took about 30 seconds to simulate a total number of 2.7 million instructions, giving an instruction rate of approximately 90,000 instructions per second.

4.1.2 Test 2: full system simulation

For a bit more of a challenge, this test boots Linux on two simulated computers – a server and a client, and runs the Netperf maerts[1] benchmark. Both computers have two instances of simple CPUs (two-way CMP).

The configuration files for full system simulation can be found in `m5/configs/fullsys/`. The `run.py` file in this directory is different than the file from the previous test in the way that it contains multiple choices of configuration, that can be selected by specifying environmental variables on the command line. These options include which benchmark to run, what kind of system to set up, etc. Environmental variables are specified with the "`-E <var>[=<val>]`" syntax (if no value, set to 'True').

The command line to start this test is:

```
ALPHA_FS/m5.opt -d output-fullsys-netperf/
  -E TEST=NETPERF_MAERTS -E NUMCPUS=2
  ../configs/fullsys/run.py
```

What happens is that two computers are simulated, both booting Linux with network drivers, so that they will be able to communicate. Then startup scripts (which are like normal Linux startup scripts – "`*.rcS`" files) start the necessary drivers, and then the benchmark programs. Figure 4.1 shows a screenshot of the simulation, after both machines are booted, and have started the benchmark programs. Notice that it is possible to interact with the simulated computer through a console interface (the application "m5term" makes that possible, and can be found in `m5/util/term/`). Interaction is slow, but working.

A similar output statistics file is produced with this test; now there are statistics for both computers, and both CPUs for each of those, but less details as the CPUs were configured as simple processors this time. Some samples are shown in Listing 4.3, while a statistics output file in its entirety can be found in Appendix A. The host instruction rate for this test is about 1.9 million instructions per second, more than one order of magnitude faster than in the previous test. This is probably because simple rather than detailed CPUs are simulated.

4.2 M5 modification experiment

The primary purpose of this experiment was to test the extensibility of M5 in practice, by attempting to add a new component to the simulator. While I

[1] Netperf is a micro-benchmark family from Hewlett-Packard for measuring bandwidth and latency characteristics of networks [Com06]. The maerts benchmark tests receiving data.

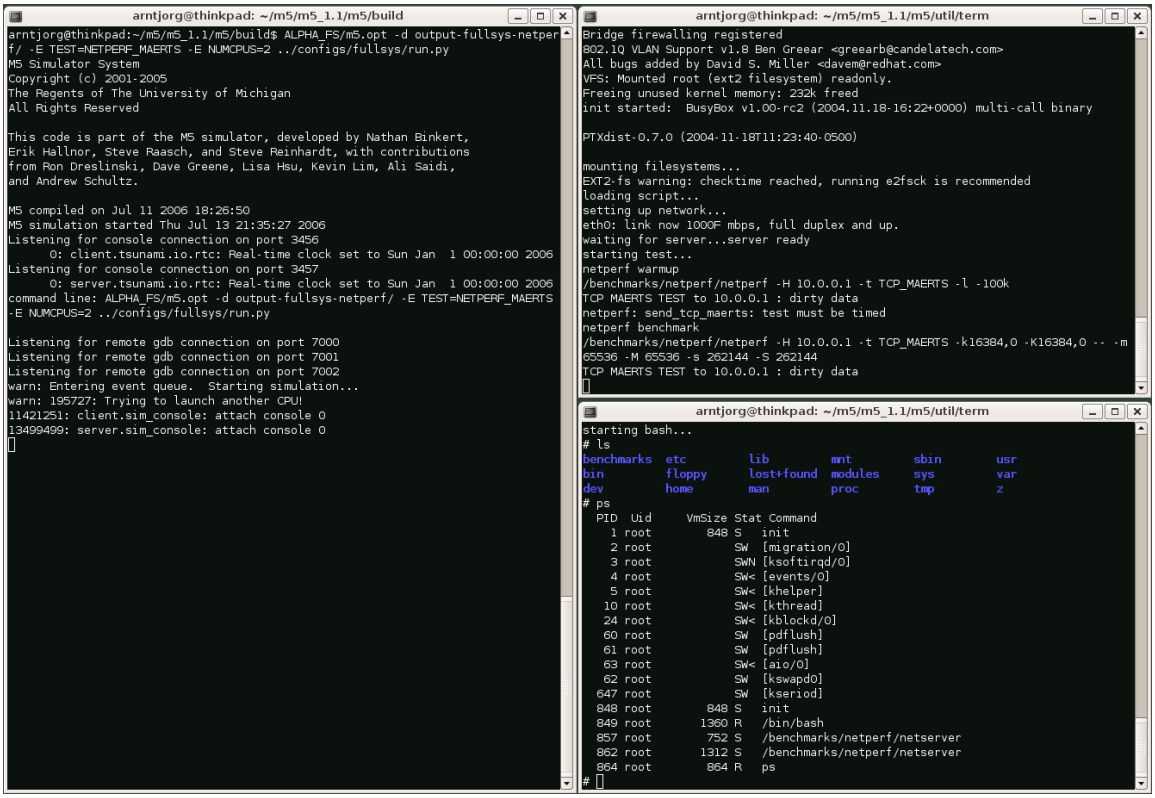

Figure 4.1: Screenshot from a full system simulation. The simulation binary is started from the left window, and the two consoles to the right are connected to the simulator, showing the console output from the client (top) and server (bottom). While the server runs the benchmark program in the background, it is possible to interact with the computer, as shown.

```
---------- Begin Simulation Statistics ----------

client.cpu0.dtb.accesses                          6685459
client.cpu0.dtb.acv                                    54
client.tsunami.etherdev0.totBandwidth          3047239733
client.tsunami.etherdev0.totBytes                50544786
client.tsunami.etherdev0.totPackets                 47633
sim_seconds                                      0.132697
sim_ticks                                       265393158
```

Listing 4.3: A small sample from the statistics output from the full system test.

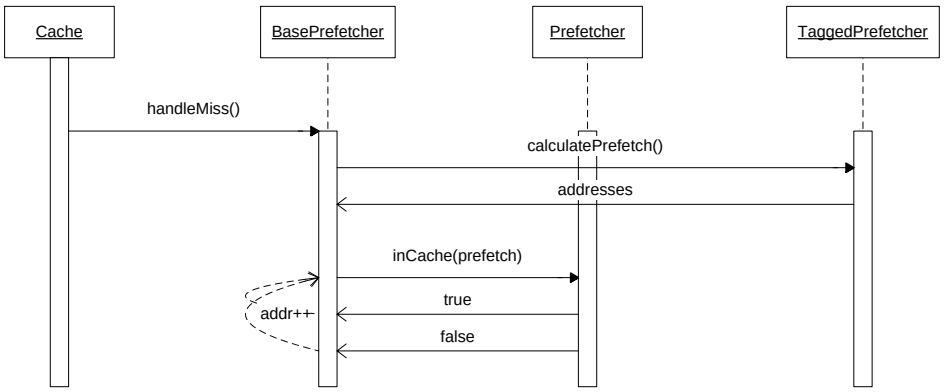

Figure 4.2: Sequence diagram for the operation of the tagged prefetcher. Many details are omitted; I show only the few calls that are relevant for the understanding of the operation. (Notation differs a bit from standard UML notation.)

did not have the time to construct a new major component from scratch, I chose to modify an existing component, add it to the simulator, and check the effects. I hope and believe the process of adding a larger and more significant new component will resemble that of a simple one, though it will probably require more work to integrate the new component into the simulator (especially customising the Python configuration files).

My target was the hardware prefetcher, described in Section 3.1.3. As it comes with the simulator, there is only one prefetcher available: a tagged prefetcher. A (simplified) sequence diagram is shown of the working of this prefetcher in Figure 4.2. It is not really correct to picture the three classes as separate objects collaborating, since `TaggedPrefetcher` inherits `Prefetcher`, which again inherits `BasePrefetcher`. I merely show them as collaborating objects to understand how the source code is distributed across those three classes.

My minor experiment was to create a dumb, sequential prefetcher that prefetches

one instruction ahead on misses, irrespective of if the address is already in cache or not. To achieve that, I changed the implementation of the virtual function `bool inCache(MemReqPtr &req)` in `base_prefetcher.hh` to always return false. It was also necessary to change the method `calculatePrefetch(...)` to only calculate one address ahead. The files can be found in Appendix B.

The simulator issued a lot warnings for the times the prefetcher prefetched an address already in cache,

```
warn: Trying to issue a prefetch to a block we already have
scabwarn: Trying to issue a prefetch to a block we already
  have
warn: Trying to issue a prefetch to a block we already have
```

Listing 4.4 shows the different in statistics output from the two simulators. Simulation is in both cases started with the command

```
ALPHA_SE/m5.opt -d output m5-test/test4/run.py
  --root.cpu.max_insts_any_thread='5000000'
  --root.cpu.l2.prefetch_policy='tagged'
  --root.cpu.l2.prefetch_miss='True'
  --root.cpu.l2.prefetch_degree='10'
```

To use the new implementation of the prefetcher, in `seq_prefetcher.hh` and `seq_prefetcher.cc`, a reference to the `.cc` file must be added to the `SConscript` file in `m5/`, to include the file in the build. Also, `m5/mem/cache/cache_builder.cc` needs to be updated with the correct file reference.

```
--------------- Old prefetcher ---------------

cpu.COM:IPC                                    4.615421
                # Committed instructions per cycle
cpu.l2.hwpf_accesses                             36212
                # number of hwpf accesses(hits+misses)
host_inst_rate                                   67482
                # Simulator instruction rate (inst/s)

--------------- New prefetcher ---------------

cpu.COM:IPC                                    4.515022
                # Committed instructions per cycle
cpu.l2.hwpf_accesses                             35129
                # number of hwpf accesses(hits+misses)
host_inst_rate                                   21987
                # Simulator instruction rate (inst/s)
```

Listing 4.4: Comparison of some key figures from a simulation with the old `TaggedPrefetecher`, and the new `SeqPrefetcher`. The numbers are similar, except for the host instruction rate, which is significantly lower (due to console output). IPC also drops a small amount.

Chapter 5

Conclusions

Chip multiprocessor architectures have recently become increasingly popular, because of several factors making it difficult to continue making faster single-threaded processor designs, such as has been done for the past decades. Power usage, heat, extreme complexity, and lack of parallelism in single-threaded programs, have made computer architects look for ways to exploit parallelism that exists between processes and applications. With the large number of transistors that can reside on a single chip today, it is possible to implement multiple processing cores on a chip, which leads to the term chip multiprocessor (CMP).

The group for computer architecture research at the Department of Computer and Information Science at NTNU does research on topics related to chip multiprocessor designs. It is thus important to have a good simulator tool available to test new ideas experimentally. A wide range of computer architecture simulators are available; a number of them are capable of simulating chip multiprocessor architectures. I have surveyed 9 different of those simulators, and taken a closer look at one of them, the simulator called M5.

The simulators have varying characteristics, strengths and weaknesses, and relevance to IDI's research. While most development projects are only a few years old, I consider a simulator such as SimOS to be too old to be used actively in research now. Asim has the problem that it is not available publicly, and would be hard to obtain. Two of the simulators – TFsim and GEMS – use a timing-first approach to simulation, where a less extensive model simulates the timing of the processor, which is then later verified by a larger and more complex execution-drive simulator, such as Simics. All the simulators can simulate CMP architectures, but only a few can simulate SMT.

My choice of simulator fell on M5. M5 is a full system simulator for the Alpha ISA (with promise about more to come), that supports CMP, SMT (partially still), and simulation of networked computers. It is open source, licensed under

GPL, and has an active development team. In my opinion, the simulator is already a well-developed product; it comes with configuration files that make things work immediately, and are a good starting point for customising the simulator. The available documentation is good, but is distributed across several locations, making it sometimes hard to find the information one is looking for. The simulator is extensible through a modular, object-oriented design, but it is unavoidable to look around in the source code a good deal if one wishes to change or make a new component.

Appendix A

M5 statistics output file

Statistics output from full system simulation

```
---------- Begin Simulation Statistics ----------
client.cpu0.dtb.accesses                      6685459
client.cpu0.dtb.acv                                54
client.cpu0.dtb.hits                           58345633
client.cpu0.dtb.misses                          2459
client.cpu0.dtb.read_accesses                  1193149
client.cpu0.dtb.read_acv                           24
client.cpu0.dtb.read_hits                      36430644
client.cpu0.dtb.read_misses                     1907
client.cpu0.dtb.write_accesses                 5492310
client.cpu0.dtb.write_acv                          30
client.cpu0.dtb.write_hits                     21914989
client.cpu0.dtb.write_misses                     552
client.cpu0.idle_fraction                     0.967426
client.cpu0.itb.accesses                       1077469
client.cpu0.itb.acv                                23
client.cpu0.itb.hits                           1076680
client.cpu0.itb.misses                           789
client.cpu0.kern.callpal                        581034
client.cpu0.kern.callpal_wripir                      4      0.00%      0.00%
client.cpu0.kern.callpal_swpctx                   7750      1.33%      1.33%
client.cpu0.kern.callpal_tbi                         2      0.00%      1.33%
client.cpu0.kern.callpal_swpipl                 326654     56.22%     57.55%
client.cpu0.kern.callpal_rdps                   200781     34.56%     92.11%
client.cpu0.kern.callpal_wrusp                       2      0.00%     92.11%
client.cpu0.kern.callpal_rdusp                       1      0.00%     92.11%
client.cpu0.kern.callpal_rti                     25005      4.30%     96.41%
client.cpu0.kern.callpal_callsys                 20805      3.58%     99.99%
client.cpu0.kern.callpal_imb                        30      0.01%    100.00%
client.cpu0.kern.faults                           7269
client.cpu0.kern.faults_interrupt                 3944     54.26%     54.26%
client.cpu0.kern.faults_dtb_miss_single           1959     26.95%     81.21%
client.cpu0.kern.faults_dtb_miss_double            500      6.88%     88.09%
client.cpu0.kern.faults_dfault                      30      0.41%     88.50%
```

client.cpu0.kern.faults_dfault	24	0.33%	88.83%
client.cpu0.kern.faults_itbmiss	789	10.85%	99.68%
client.cpu0.kern.faults_iaccvio	23	0.32%	100.00%
client.cpu0.kern.inst.arm	0		
client.cpu0.kern.inst.hwrei	588303		
client.cpu0.kern.inst.ivlb	0		
client.cpu0.kern.inst.ivle	0		
client.cpu0.kern.inst.quiesce	3799		
client.cpu0.kern.ipl_count	355603		
client.cpu0.kern.ipl_count_0	167717	47.16%	47.16%
client.cpu0.kern.ipl_count_21	42	0.01%	47.18%
client.cpu0.kern.ipl_count_22	385	0.11%	47.28%
client.cpu0.kern.ipl_count_30	3766	1.06%	48.34%
client.cpu0.kern.ipl_count_31	183693	51.66%	100.00%
client.cpu0.kern.ipl_good	336102		
client.cpu0.kern.ipl_good_0	167713	49.90%	49.90%
client.cpu0.kern.ipl_good_21	42	0.01%	49.91%
client.cpu0.kern.ipl_good_22	385	0.11%	50.03%
client.cpu0.kern.ipl_good_30	3766	1.12%	51.15%
client.cpu0.kern.ipl_good_31	164196	48.85%	100.00%
client.cpu0.kern.ipl_ticks	265909335		
client.cpu0.kern.ipl_ticks_0	246012985	92.52%	92.52%
client.cpu0.kern.ipl_ticks_21	7098	0.00%	92.52%
client.cpu0.kern.ipl_ticks_22	128208	0.05%	92.57%
client.cpu0.kern.ipl_ticks_30	851152	0.32%	92.89%
client.cpu0.kern.ipl_ticks_31	18909892	7.11%	100.00%
client.cpu0.kern.ipl_used	0.945161		
client.cpu0.kern.ipl_used_0	0.999976		
client.cpu0.kern.ipl_used_21	1		
client.cpu0.kern.ipl_used_22	1		
client.cpu0.kern.ipl_used_30	1		
client.cpu0.kern.ipl_used_31	0.893861		
client.cpu0.kern.mode_good_kernel	24822		
client.cpu0.kern.mode_good_user	21057		
client.cpu0.kern.mode_good_idle	3764		
client.cpu0.kern.mode_good_interrupt	0		
client.cpu0.kern.mode_switch_kernel	46411		
client.cpu0.kern.mode_switch_user	21057		
client.cpu0.kern.mode_switch_idle	11528		
client.cpu0.kern.mode_switch_interrupt	0		
client.cpu0.kern.mode_switch_good	0.628424		
client.cpu0.kern.mode_switch_good_kernel	0.534830		
client.cpu0.kern.mode_switch_good_user	1		
client.cpu0.kern.mode_switch_good_idle	0.326509		
client.cpu0.kern.mode_switch_good_interrupt	<err: div-0>		
client.cpu0.kern.mode_ticks_kernel	185868608	69.88%	69.88%
client.cpu0.kern.mode_ticks_user	1340605	0.50%	70.38%
client.cpu0.kern.mode_ticks_idle	78771127	29.62%	100.00%
client.cpu0.kern.mode_ticks_interrupt	0	0.00%	100.00%
client.cpu0.kern.swap_context	7750		
client.cpu0.kern.syscall	20791		
client.cpu0.kern.syscall_fork	1	0.00%	0.00%
client.cpu0.kern.syscall_read	4	0.02%	0.02%
client.cpu0.kern.syscall_close	4	0.02%	0.04%

client.cpu0.kern.syscall_obreak	6	0.03%	0.07%
client.cpu0.kern.syscall_lseek	1	0.00%	0.08%
client.cpu0.kern.syscall_open	9	0.04%	0.12%
client.cpu0.kern.syscall_sigprocmask	1	0.00%	0.13%
client.cpu0.kern.syscall_execve	1	0.00%	0.13%
client.cpu0.kern.syscall_pre_F64_stat	2	0.01%	0.14%
client.cpu0.kern.syscall_mmap	10	0.05%	0.19%
client.cpu0.kern.syscall_mprotect	3	0.01%	0.20%
client.cpu0.kern.syscall_pre_F64_fstat	3	0.01%	0.22%
client.cpu0.kern.syscall_socket	1	0.00%	0.22%
client.cpu0.kern.syscall_connect	1	0.00%	0.23%
client.cpu0.kern.syscall_old_recv	20744	99.77%	100.00%
client.cpu0.not_idle_fraction	0.032574		
client.cpu0.numCycles	190952626		
client.cpu0.num_insts	190951814		
client.cpu0.num_refs	58605345		
client.cpu1.dtb.accesses	3718122		
client.cpu1.dtb.acv	86		
client.cpu1.dtb.hits	71322254		
client.cpu1.dtb.misses	2875		
client.cpu1.dtb.read_accesses	2643831		
client.cpu1.dtb.read_acv	49		
client.cpu1.dtb.read_hits	48641930		
client.cpu1.dtb.read_misses	1964		
client.cpu1.dtb.write_accesses	1074291		
client.cpu1.dtb.write_acv	37		
client.cpu1.dtb.write_hits	22680324		
client.cpu1.dtb.write_misses	911		
client.cpu1.idle_fraction	0.955019		
client.cpu1.itb.accesses	2004537		
client.cpu1.itb.acv	46		
client.cpu1.itb.hits	2003661		
client.cpu1.itb.misses	876		
client.cpu1.kern.callpal	866280		
client.cpu1.kern.callpal_wripir	3766	0.43%	0.43%
client.cpu1.kern.callpal_swpctx	245	0.03%	0.46%
client.cpu1.kern.callpal_tbi	4	0.00%	0.46%
client.cpu1.kern.callpal_swpipl	643686	74.30%	74.77%
client.cpu1.kern.callpal_rdps	202622	23.39%	98.16%
client.cpu1.kern.callpal_wrusp	1	0.00%	98.16%
client.cpu1.kern.callpal_rdusp	2	0.00%	98.16%
client.cpu1.kern.callpal_rti	15746	1.82%	99.98%
client.cpu1.kern.callpal_callsys	171	0.02%	100.00%
client.cpu1.kern.callpal_imb	37	0.00%	100.00%
client.cpu1.kern.faults	19135		
client.cpu1.kern.faults_interrupt	15252	79.71%	79.71%
client.cpu1.kern.faults_dtb_miss_single	2302	12.03%	91.74%
client.cpu1.kern.faults_dtb_miss_double	573	2.99%	94.73%
client.cpu1.kern.faults_dfault	37	0.19%	94.93%
client.cpu1.kern.faults_dfault	49	0.26%	95.18%
client.cpu1.kern.faults_itbmiss	876	4.58%	99.76%
client.cpu1.kern.faults_iaccvio	46	0.24%	100.00%
client.cpu1.kern.inst.arm	0		
client.cpu1.kern.inst.hwrei	885415		

client.cpu1.kern.inst.ivlb	0		
client.cpu1.kern.inst.ivle	0		
client.cpu1.kern.inst.quiesce	8		
client.cpu1.kern.ipl_count	674684		
client.cpu1.kern.ipl_count_0	200342	29.69%	29.69%
client.cpu1.kern.ipl_count_21	15113	2.24%	31.93%
client.cpu1.kern.ipl_count_22	282	0.04%	31.98%
client.cpu1.kern.ipl_count_30	4	0.00%	31.98%
client.cpu1.kern.ipl_count_31	458943	68.02%	100.00%
client.cpu1.kern.ipl_good	415989		
client.cpu1.kern.ipl_good_0	200291	48.15%	48.15%
client.cpu1.kern.ipl_good_21	15113	3.63%	51.78%
client.cpu1.kern.ipl_good_22	282	0.07%	51.85%
client.cpu1.kern.ipl_good_30	4	0.00%	51.85%
client.cpu1.kern.ipl_good_31	200299	48.15%	100.00%
client.cpu1.kern.ipl_ticks	265393564		
client.cpu1.kern.ipl_ticks_0	206432363	77.78%	77.78%
client.cpu1.kern.ipl_ticks_21	2448175	0.92%	78.71%
client.cpu1.kern.ipl_ticks_22	92406	0.03%	78.74%
client.cpu1.kern.ipl_ticks_30	976	0.00%	78.74%
client.cpu1.kern.ipl_ticks_31	56419644	21.26%	100.00%
client.cpu1.kern.ipl_used	0.616569		
client.cpu1.kern.ipl_used_0	0.999745		
client.cpu1.kern.ipl_used_21	1		
client.cpu1.kern.ipl_used_22	1		
client.cpu1.kern.ipl_used_30	1		
client.cpu1.kern.ipl_used_31	0.436435		
client.cpu1.kern.mode_good_kernel	15559		
client.cpu1.kern.mode_good_user	446		
client.cpu1.kern.mode_good_idle	0		
client.cpu1.kern.mode_good_interrupt	15112		
client.cpu1.kern.mode_switch_kernel	31891		
client.cpu1.kern.mode_switch_user	446		
client.cpu1.kern.mode_switch_idle	0		
client.cpu1.kern.mode_switch_interrupt	15112		
client.cpu1.kern.mode_switch_good	0.655799		
client.cpu1.kern.mode_switch_good_kernel	0.487881		
client.cpu1.kern.mode_switch_good_user	1		
client.cpu1.kern.mode_switch_good_idle	no value		
client.cpu1.kern.mode_switch_good_interrupt	1		
client.cpu1.kern.mode_ticks_kernel	233554109	88.00%	88.00%
client.cpu1.kern.mode_ticks_user	2065910	0.78%	88.78%
client.cpu1.kern.mode_ticks_idle	0	0.00%	88.78%
client.cpu1.kern.mode_ticks_interrupt	29768024	11.22%	100.00%
client.cpu1.kern.swap_context	245		
client.cpu1.kern.syscall	142		
client.cpu1.kern.syscall_fork	2	1.41%	1.41%
client.cpu1.kern.syscall_read	3	2.11%	3.52%
client.cpu1.kern.syscall_write	1	0.70%	4.23%
client.cpu1.kern.syscall_close	3	2.11%	6.34%
client.cpu1.kern.syscall_obreak	19	13.38%	19.72%
client.cpu1.kern.syscall_lseek	1	0.70%	20.42%
client.cpu1.kern.syscall_getpid	1	0.70%	21.13%
client.cpu1.kern.syscall_open	4	2.82%	23.94%

client.cpu1.kern.syscall_sigprocmask	2	1.41%	25.35%
client.cpu1.kern.syscall_ioctl	1	0.70%	26.06%
client.cpu1.kern.syscall_execve	2	1.41%	27.46%
client.cpu1.kern.syscall_pre_F64_stat	1	0.70%	28.17%
client.cpu1.kern.syscall_mmap	5	3.52%	31.69%
client.cpu1.kern.syscall_mprotect	1	0.70%	32.39%
client.cpu1.kern.syscall_pre_F64_fstat	2	1.41%	33.80%
client.cpu1.kern.syscall_select	2	1.41%	35.21%
client.cpu1.kern.syscall_socket	1	0.70%	35.92%
client.cpu1.kern.syscall_connect	1	0.70%	36.62%
client.cpu1.kern.syscall_old_send	2	1.41%	38.03%
client.cpu1.kern.syscall_old_recv	82	57.75%	95.77%
client.cpu1.kern.syscall_bind	1	0.70%	96.48%
client.cpu1.kern.syscall_setsockopt	3	2.11%	98.59%
client.cpu1.kern.syscall_getsockopt	2	1.41%	100.00%
client.cpu1.not_idle_fraction	0.044981		
client.cpu1.numCycles	263684542		
client.cpu1.num_insts	263683620		
client.cpu1.num_refs	71918107		
client.kern.fnCalls	0		
client.tsunami.etherdev0.coalescedRxDesc	0		
client.tsunami.etherdev0.coalescedRxIdle	0		
client.tsunami.etherdev0.coalescedRxOk	0		
client.tsunami.etherdev0.coalescedRxOrn	0		
client.tsunami.etherdev0.coalescedSwi	0		
client.tsunami.etherdev0.coalescedTotal	1		
client.tsunami.etherdev0.coalescedTxDesc	0		
client.tsunami.etherdev0.coalescedTxIdle	1		
client.tsunami.etherdev0.coalescedTxOk	0		
client.tsunami.etherdev0.descDMAReads	14699		
client.tsunami.etherdev0.descDMAWrites	47633		
client.tsunami.etherdev0.descDmaReadBytes	235184		
client.tsunami.etherdev0.descDmaWriteBytes	381064		
client.tsunami.etherdev0.droppedPackets	0		
client.tsunami.etherdev0.postedInterrupts	15112		
client.tsunami.etherdev0.postedRxDesc	467		
client.tsunami.etherdev0.postedRxIdle	0		
client.tsunami.etherdev0.postedRxOk	0		
client.tsunami.etherdev0.postedRxOrn	0		
client.tsunami.etherdev0.postedSwi	0		
client.tsunami.etherdev0.postedTxDesc	0		
client.tsunami.etherdev0.postedTxIdle	14645		
client.tsunami.etherdev0.postedTxOk	1		
client.tsunami.etherdev0.rxBandwidth	2999354610		
client.tsunami.etherdev0.rxBytes	49750512		
client.tsunami.etherdev0.rxIpChecksums	32934		
client.tsunami.etherdev0.rxPPS	248190		
client.tsunami.etherdev0.rxPackets	32934		
client.tsunami.etherdev0.rxTcpChecksums	32934		
client.tsunami.etherdev0.rxUdpChecksums	0		
client.tsunami.etherdev0.totBandwidth	3047239733		
client.tsunami.etherdev0.totBytes	50544786		
client.tsunami.etherdev0.totPackets	47633		
client.tsunami.etherdev0.totalRxDesc	1077		

client.tsunami.etherdev0.totalRxIdle	0		
client.tsunami.etherdev0.totalRxOk	0		
client.tsunami.etherdev0.totalRxOrn	0		
client.tsunami.etherdev0.totalSwi	0		
client.tsunami.etherdev0.totalTxDesc	0		
client.tsunami.etherdev0.totalTxIdle	14699		
client.tsunami.etherdev0.totalTxOk	0		
client.tsunami.etherdev0.txBandwidth	47885123		
client.tsunami.etherdev0.txBytes	794274		
client.tsunami.etherdev0.txIpChecksums	2		
client.tsunami.etherdev0.txPPS	110772		
client.tsunami.etherdev0.txPackets	14699		
client.tsunami.etherdev0.txTcpChecksums	2		
client.tsunami.etherdev0.txUdpChecksums	0		
host_inst_rate	1905131		
host_mem_usage	352108		
host_seconds	997.57		
host_tick_rate	266039		
server.cpu.dtb.accesses	322632589		
server.cpu.dtb.acv	219		
server.cpu.dtb.hits	444851646		
server.cpu.dtb.misses	15435		
server.cpu.dtb.read_accesses	213960701		
server.cpu.dtb.read_acv	102		
server.cpu.dtb.read_hits	287993853		
server.cpu.dtb.read_misses	9348		
server.cpu.dtb.write_accesses	108671888		
server.cpu.dtb.write_acv	117		
server.cpu.dtb.write_hits	156857793		
server.cpu.dtb.write_misses	6087		
server.cpu.idle_fraction	0.969169		
server.cpu.itb.accesses	999249599		
server.cpu.itb.acv	90		
server.cpu.itb.hits	999244725		
server.cpu.itb.misses	4874		
server.cpu.kern.callpal	2263614		
server.cpu.kern.callpal_swpctx	981	0.04%	0.04%
server.cpu.kern.callpal_tbi	52	0.00%	0.05%
server.cpu.kern.callpal_swpipl	1855213	81.96%	82.00%
server.cpu.kern.callpal_rdps	353496	15.62%	97.62%
server.cpu.kern.callpal_wrusp	3	0.00%	97.62%
server.cpu.kern.callpal_rdusp	6	0.00%	97.62%
server.cpu.kern.callpal_rti	52008	2.30%	99.92%
server.cpu.kern.callpal_callsys	1701	0.08%	99.99%
server.cpu.kern.callpal_imb	154	0.01%	100.00%
server.cpu.kern.faults	70137		
server.cpu.kern.faults_interrupt	49519	70.60%	70.60%
server.cpu.kern.faults_dtb_miss_single	13290	18.95%	89.55%
server.cpu.kern.faults_dtb_miss_double	2145	3.06%	92.61%
server.cpu.kern.faults_dfault	117	0.17%	92.78%
server.cpu.kern.faults_dfault	102	0.15%	92.92%
server.cpu.kern.faults_itbmiss	4874	6.95%	99.87%
server.cpu.kern.faults_iaccvio	90	0.13%	100.00%
server.cpu.kern.inst.arm	0		

server.cpu.kern.inst.hwrei	2333751		
server.cpu.kern.inst.ivlb	0		
server.cpu.kern.inst.ivle	0		
server.cpu.kern.inst.quiesce	4784		
server.cpu.kern.ipl_count	1956740		
server.cpu.kern.ipl_count_0	782711	40.00%	40.00%
server.cpu.kern.ipl_count_21	49383	2.52%	42.52%
server.cpu.kern.ipl_count_22	401	0.02%	42.54%
server.cpu.kern.ipl_count_31	1124245	57.46%	100.00%
server.cpu.kern.ipl_good	1615219		
server.cpu.kern.ipl_good_0	782585	48.45%	48.45%
server.cpu.kern.ipl_good_21	49383	3.06%	51.51%
server.cpu.kern.ipl_good_22	401	0.02%	51.53%
server.cpu.kern.ipl_good_31	782850	48.47%	100.00%
server.cpu.kern.ipl_ticks	266403576		
server.cpu.kern.ipl_ticks_0	255405697	95.87%	95.87%
server.cpu.kern.ipl_ticks_21	997264	0.37%	96.25%
server.cpu.kern.ipl_ticks_22	14133	0.01%	96.25%
server.cpu.kern.ipl_ticks_31	9986482	3.75%	100.00%
server.cpu.kern.ipl_used	0.825464		
server.cpu.kern.ipl_used_0	0.999839		
server.cpu.kern.ipl_used_21	1		
server.cpu.kern.ipl_used_22	1		
server.cpu.kern.ipl_used_31	0.696334		
server.cpu.kern.mode_good_kernel	52270		
server.cpu.kern.mode_good_user	10685		
server.cpu.kern.mode_good_idle	137		
server.cpu.kern.mode_good_interrupt	41448		
server.cpu.kern.mode_switch_kernel	128343		
server.cpu.kern.mode_switch_user	10685		
server.cpu.kern.mode_switch_idle	28815		
server.cpu.kern.mode_switch_interrupt	41448		
server.cpu.kern.mode_switch_good	0.499496		
server.cpu.kern.mode_switch_good_kernel	0.407268		
server.cpu.kern.mode_switch_good_user	1		
server.cpu.kern.mode_switch_good_idle	0.004754		
server.cpu.kern.mode_switch_good_interrupt	1		
server.cpu.kern.mode_ticks_kernel	42662207	16.01%	16.01%
server.cpu.kern.mode_ticks_user	124955038	46.90%	62.92%
server.cpu.kern.mode_ticks_idle	93934709	35.26%	98.18%
server.cpu.kern.mode_ticks_interrupt	4851716	1.82%	100.00%
server.cpu.kern.swap_context	981		
server.cpu.kern.syscall	1646		
server.cpu.kern.syscall_fork	4	0.24%	0.24%
server.cpu.kern.syscall_read	123	7.47%	7.72%
server.cpu.kern.syscall_write	42	2.55%	10.27%
server.cpu.kern.syscall_close	114	6.93%	17.19%
server.cpu.kern.syscall_obreak	27	1.64%	18.83%
server.cpu.kern.syscall_setuid	3	0.18%	19.02%
server.cpu.kern.syscall_getuid	3	0.18%	19.20%
server.cpu.kern.syscall_open	124	7.53%	26.73%
server.cpu.kern.syscall_getgid	3	0.18%	26.91%
server.cpu.kern.syscall_sigprocmask	3	0.18%	27.10%
server.cpu.kern.syscall_ioctl	11	0.67%	27.76%

server.cpu.kern.syscall_execve	3	0.18%	27.95%
server.cpu.kern.syscall_pre_F64_stat	45	2.73%	30.68%
server.cpu.kern.syscall_pre_F64_lstat	37	2.25%	32.93%
server.cpu.kern.syscall_mmap	126	7.65%	40.58%
server.cpu.kern.syscall_munmap	105	6.38%	46.96%
server.cpu.kern.syscall_mprotect	6	0.36%	47.33%
server.cpu.kern.syscall_pre_F64_fstat	114	6.93%	54.25%
server.cpu.kern.syscall_fcntl	3	0.18%	54.43%
server.cpu.kern.syscall_socket	1	0.06%	54.50%
server.cpu.kern.syscall_old_accept	2	0.12%	54.62%
server.cpu.kern.syscall_old_send	734	44.59%	99.21%
server.cpu.kern.syscall_old_recv	2	0.12%	99.33%
server.cpu.kern.syscall_bind	1	0.06%	99.39%
server.cpu.kern.syscall_setsockopt	3	0.18%	99.57%
server.cpu.kern.syscall_listen	1	0.06%	99.64%
server.cpu.kern.syscall_getsockopt	2	0.12%	99.76%
server.cpu.kern.syscall_setgid	3	0.18%	99.94%
server.cpu.kern.syscall_old_getsockname	1	0.06%	100.00%
server.cpu.not_idle_fraction	0.030831		
server.cpu.numCycles	1445878294		
server.cpu.num_insts	1445873330		
server.cpu.num_refs	445624373		
server.kern.fnCalls	0		
server.tsunami.etherdev0.coalescedRxDesc	0		
server.tsunami.etherdev0.coalescedRxIdle	0		
server.tsunami.etherdev0.coalescedRxOk	0		
server.tsunami.etherdev0.coalescedRxOrn	0		
server.tsunami.etherdev0.coalescedSwi	0		
server.tsunami.etherdev0.coalescedTotal	1		
server.tsunami.etherdev0.coalescedTxDesc	0		
server.tsunami.etherdev0.coalescedTxIdle	1		
server.tsunami.etherdev0.coalescedTxOk	0		
server.tsunami.etherdev0.descDMAReads	32934		
server.tsunami.etherdev0.descDMAWrites	47633		
server.tsunami.etherdev0.descDmaReadBytes	526944		
server.tsunami.etherdev0.descDmaWriteBytes	381064		
server.tsunami.etherdev0.droppedPackets	0		
server.tsunami.etherdev0.postedInterrupts	47531		
server.tsunami.etherdev0.postedRxDesc	14688		
server.tsunami.etherdev0.postedRxIdle	0		
server.tsunami.etherdev0.postedRxOk	0		
server.tsunami.etherdev0.postedRxOrn	0		
server.tsunami.etherdev0.postedSwi	0		
server.tsunami.etherdev0.postedTxDesc	0		
server.tsunami.etherdev0.postedTxIdle	32934		
server.tsunami.etherdev0.postedTxOk	0		
server.tsunami.etherdev0.rxBandwidth	47885123		
server.tsunami.etherdev0.rxBytes	794274		
server.tsunami.etherdev0.rxIpChecksums	14699		
server.tsunami.etherdev0.rxPPS	110772		
server.tsunami.etherdev0.rxPackets	14699		
server.tsunami.etherdev0.rxTcpChecksums	14699		
server.tsunami.etherdev0.rxUdpChecksums	0		
server.tsunami.etherdev0.totBandwidth	3047239733		

```
server.tsunami.etherdev0.totBytes            50544786
server.tsunami.etherdev0.totPackets             47633
server.tsunami.etherdev0.totalRxDesc            14686
server.tsunami.etherdev0.totalRxIdle                0
server.tsunami.etherdev0.totalRxOk                  0
server.tsunami.etherdev0.totalRxOrn                 0
server.tsunami.etherdev0.totalSwi                   0
server.tsunami.etherdev0.totalTxDesc                0
server.tsunami.etherdev0.totalTxIdle            32934
server.tsunami.etherdev0.totalTxOk                  0
server.tsunami.etherdev0.txBandwidth       2999354610
server.tsunami.etherdev0.txBytes             49750512
server.tsunami.etherdev0.txIpChecksums          32931
server.tsunami.etherdev0.txPPS                 248190
server.tsunami.etherdev0.txPackets              32934
server.tsunami.etherdev0.txTcpChecksums         32931
server.tsunami.etherdev0.txUdpChecksums             0
sim_freq                                   2000000000
sim_insts                                  1900508764
sim_seconds                                  0.132697
sim_ticks                                   265393158

---------- End Simulation Statistics   ----------


Statistics output from full system simulation
```

Appendix B

Prefetcher files

Listing B.1: seq_prefetcher.hh

```
1
2  /*
3   * Copyright (c) 2005
4   * The Regents of The University of Michigan
5   * All Rights Reserved
6   *
7   * This code is part of the M5 simulator, developed by Nathan
      Binkert,
8   * Erik Hallnor, Steve Raasch, and Steve Reinhardt, with
      contributions
9   * from Ron Dreslinski, Dave Greene, Lisa Hsu, Kevin Lim, Ali
      Saidi,
10  * and Andrew Schultz.
11  *
12  * Permission is granted to use, copy, create derivative works
      and
13  * redistribute this software and such derivative works for any
14  * purpose, so long as the copyright notice above, this grant
      of
15  * permission, and the disclaimer below appear in all copies
      made; and
16  * so long as the name of The University of Michigan is not
      used in
17  * any advertising or publicity pertaining to the use or
      distribution
18  * of this software without specific, written prior
      authorization.
19  *
20  * THIS SOFTWARE IS PROVIDED AS IS, WITHOUT REPRESENTATION FROM
      THE
21  * UNIVERSITY OF MICHIGAN AS TO ITS FITNESS FOR ANY PURPOSE,
      AND
22  * WITHOUT WARRANTY BY THE UNIVERSITY OF MICHIGAN OF ANY KIND,
      EITHER
```

```
23  * EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED
24  * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
25  * PURPOSE. THE REGENTS OF THE UNIVERSITY OF MICHIGAN SHALL NOT
        BE
26  * LIABLE FOR ANY DAMAGES, INCLUDING DIRECT, SPECIAL, INDIRECT,
27  * INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WITH RESPECT TO ANY
        CLAIM
28  * ARISING OUT OF OR IN CONNECTION WITH THE USE OF THE SOFTWARE
        , EVEN
29  * IF IT HAS BEEN OR IS HEREAFTER ADVISED OF THE POSSIBILITY OF
        SUCH
30  * DAMAGES.
31  */
32
33 /**
34  * @file
35  * Describes a tagged prefetcher based on template policies.
36  */
37
38 #ifndef __MEM_CACHE_PREFETCH_PREFETCHER_HH__
39 #define __MEM_CACHE_PREFETCH_PREFETCHER_HH__
40
41 #include "base/misc.hh" // fatal, panic, and warn
42
43 #include "mem/cache/prefetch/base_prefetcher.hh"
44
45 /**
46  * A template-policy based cache. The behavior of the cache can
        be altered by
47  * supplying different template policies. TagStore handles all
        tag and data
48  * storage @sa TagStore. Buffering handles all misses and
        writes/writebacks
49  * @sa MissQueue. Coherence handles all coherence policy
        details @sa
50  * UniCoherence, SimpleMultiCoherence.
51  */
52 class SeqPrefetcher : public BasePrefetcher
53 {
54
55   public:
56
57     SeqPrefetcher(int size, bool pageStop, bool serialSquash,
58         bool cacheCheckPush, bool onlyData)
59   :BasePrefetcher(size, pageStop, serialSquash,
60       cacheCheckPush, onlyData);
61
62     ~SeqPrefetcher() {}
63
64     void calculatePrefetch(MemReqPtr &req, std::list<Addr> &
          addresses,
65         std::list<Tick> &delays);
66
67     bool inCache(MemReqPtr &req);
```

```
68
69      bool inMissQueue(Addr address, int asid);
70
71
72  };
73
74  #endif // __MEM_CACHE_PREFETCH_PREFETCHER_HH__
```

Listing B.1: seq_prefetcher.hh

Listing B.2: seq_prefetcher.cc

```
1
2  /*
3   * Copyright (c) 2005
4   * The Regents of The University of Michigan
5   * All Rights Reserved
6   *
7   * This code is part of the M5 simulator, developed by Nathan
      Binkert,
8   * Erik Hallnor, Steve Raasch, and Steve Reinhardt, with
      contributions
9   * from Ron Dreslinski, Dave Greene, Lisa Hsu, Kevin Lim, Ali
      Saidi,
10  * and Andrew Schultz.
11  *
12  * Permission is granted to use, copy, create derivative works
      and
13  * redistribute this software and such derivative works for any
14  * purpose, so long as the copyright notice above, this grant
      of
15  * permission, and the disclaimer below appear in all copies
      made; and
16  * so long as the name of The University of Michigan is not
      used in
17  * any advertising or publicity pertaining to the use or
      distribution
18  * of this software without specific, written prior
      authorization.
19  *
20  * THIS SOFTWARE IS PROVIDED AS IS, WITHOUT REPRESENTATION FROM
      THE
21  * UNIVERSITY OF MICHIGAN AS TO ITS FITNESS FOR ANY PURPOSE,
      AND
22  * WITHOUT WARRANTY BY THE UNIVERSITY OF MICHIGAN OF ANY KIND,
      EITHER
23  * EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION THE IMPLIED
24  * WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
25  * PURPOSE. THE REGENTS OF THE UNIVERSITY OF MICHIGAN SHALL NOT
      BE
26  * LIABLE FOR ANY DAMAGES, INCLUDING DIRECT, SPECIAL, INDIRECT,
27  * INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WITH RESPECT TO ANY
      CLAIM
```

```
28   * ARISING OUT OF OR IN CONNECTION WITH THE USE OF THE SOFTWARE
       , EVEN
29   * IF IT HAS BEEN OR IS HEREAFTER ADVISED OF THE POSSIBILITY OF
       SUCH
30   * DAMAGES.
31   */
32
33  /**
34   * @file
35   * Prefetcher template instantiations.
36   */
37
38  #include "mem/cache/tags/cache_tags.hh"
39
40  #include "mem/cache/tags/lru.hh"
41
42  #include "base/compression/null_compression.hh"
43
44  #include "mem/cache/miss/miss_queue.hh"
45  #include "mem/cache/miss/blocking_buffer.hh"
46
47  #include "mem/cache/prefetch/seq_prefetcher.hh"
48
49
50      SeqPrefetcher(int size, bool pageStop, bool serialSquash,
51              bool cacheCheckPush, bool onlyData)
52      :BasePrefetcher(size, pageStop, serialSquash,
53        cacheCheckPush, onlyData)
54      {}
55
56      ~Prefetcher() {}
57
58      void calculatePrefetch(MemReqPtr &req, std::list<Addr> &
          addresses,
59            std::list<Tick> &delays)
60      {
61    Addr blkAddr = req->paddr & ~(Addr)(this->blkSize-1);
62
63        //Prefetch one address ahead
64        Addr newAddr = blkAddr + this->blkSize;
65        if (this->pageStop &&
66      (blkAddr & ~(TheISA::VMPageSize - 1)) !=
67      (newAddr & ~(TheISA::VMPageSize - 1)))
68        {
69      //Spanned the page, so now stop
70      this->pfSpanPage += degree - d + 1;
71      return;
72        }
73        else
74        {
75      addresses.push_back(newAddr);
76      delays.push_back(latency);
77        }
78
```

```
79      }
80
81      bool inCache(MemReqPtr &req)
82      {
83
84    return false;
85      }
86
87      bool inMissQueue(Addr address, int asid)
88      {
89
90    return false;
91      }
```

Listing B.2: seq_prefetcher.cc

Bibliography

[ALE02] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, February 2002.

[AMD05] Advanced Micro Devices, Inc. *Software Optimization Guide for AMD64 Processors*, 3.06 edition, September 2005. Publication # 25112. Available at: `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF`.

[BDH+05] Nathan Binkert, Ron Dreslinski, Lisa Hsu, Kevin Lim, Ali Saidi, and Steve Reinhardt. Using the M5 simulator. Tutorial at the *32nd Annual International Symposium on Computer Architecture (ISCA-32)*, June 2005. Available from: `http://m5.eecs.umich.edu`.

[BHR03] N. Binkert, E. Hallnor, and S. Reinhardt. Network-oriented full-system simulation using M5. In *Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*. February 2003.

[Com06] Hewlett-Packard Company. Netperf: A network performance benchmark. Benchmark's webpage, July 2006. Available at: `http://www.netperf.org`.

[EAB+02] J. Emer, P. Ahuja, E. Borch, A. Klauser, Chi-Keung Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: a performance model framework. *Computer*, 35(2):68–76, February 2002.

[HPRA02] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. Rsim: simulating shared-memory multiprocessors with ILP processors. *Computer*, 35(2):40–49, February 2002.

[HS00] Timothy H. Heil and James E. Smith. Relational profiling: enabling thread-level parallelism in virtual machines. In *International Symposium on Microarchitecture*, pages 281–290, 2000.

[HSW+04] Nikolaos Hardavellas, Stephen Somogyi, Thomas F. Wenisch, Roland E. Wunderlich, Shelley Chen, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. SimFlex: a fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture. *SIGMETRICS Perform. Eval. Rev.*, 31(4):31–34, 2004.

[KAO05] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, March–April 2005.

[Kni02] Steven Knight. SCons design and implementation. In *Tenth International Python Conference*, February 2002. Available at: `http://www.python10.org/p10-papers/16/index.htm`.

[Lan05] Arnt Jørgen Lande. Multithreading in chip multiprocessors. Project report from the course "Computer Design and Architecture, Specialization" at the Norwegian University of Science and Technology. Supervised by Prof. Lasse Natvig, Dept. of Computer and Information Science, December 2005.

[MAA+02] S. S. Mukherjee, S. V. Adve, T. Austin, J. Emer, and P. S. Magnusson. Performance simulation tools. *Computer*, 35(2):38–39, February 2002.

[MCE+02] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, February 2002.

[MHW02] 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, 2002. ACM Press.

[Mic06] The M5 simulator system. Simulator's webpage, July 2006. Available at `http://m5.eecs.umich.edu`.

[MSB+05] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[OH05] Kunle Olukotun and Lance Hammond. The future of microprocessors. *ACM Queue*, 3(7), September 2005.

[QTY05] Kelly Quinn, Vernon Turner, and Jessica Yang. The next evolution in enterprise computing: The convergence of multicore x86 processing and 64-bit operating systems. White paper 05C4442, IDC, Sponsored by: Advanced Micro Devices Inc., April 2005. Available at: `http://multicore.amd.com/WhitePapers/IDC_WhitePaper_Convergence_en.pdf`.

[RBS+06] Steve Reinhardt, Nathan Binkert, Ali Saidi, Ron Dreslinski, and Kevin Lim. Using the M5 simulator. Tutorial at the *33rd Annual International Symposium on Computer Architecture (ISCA-33)*, June 2006. Available from: `http://m5.eecs.umich.edu`.

[RHWG95] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: the SimOS approach. *Parallel & Distributed Technology: Systems & Applications, IEEE [see also IEEE Concurrency]*, 3(4):34–43, Winter 1995.

[SA05] L. Spracklen and S. G. Abraham. Chip multithreading: opportunities and challenges. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 248–252, February 2005.

[URŠ03] Theo Ungerer, Borut Robič, and Jurij Šilc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35(1):29–63, 2003.

[vH06] Dimitri van Heesch. Doxygen. Project's homepage, May 2006. Available at: `http://www.doxygen.org`.

[Wal91] David W. Wall. Limits of instruction-level parallelism. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 176–188, New York, NY, USA, 1991. ACM Press.

[Wik06] Moore's law. From Wikipedia, the free encyclopedia, July 2006. Available at: `http://en.wikipedia.org/wiki/Moore's_law`.

[WL97] S. P. Vander Wiel and D. J. Lilja. When caches aren't enough: data prefetching techniques. *Computer*, 30(7):23–30, July 1997.

[WW05] Thomas F. Wenisch and Roland E. Wunderlich. SimFlex: Fast, accurate and flexible simulation of computer systems. Tutorial at the *International Symposium on Microarchitecture (MICRO-38)*, November 2005. Available at `http://www.ece.cmu.edu/~simflex/software/SimFlex-tutorial.pdf`.