Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors

R. Ubal, J. Sahuquillo, S. Petit and P. López
Universidad Politécnica de Valencia
Camino de Vera s/n 46021 Valencia, Spain
raurte@gap.upv.es

Abstract

Current microprocessors are based in complex designs, integrating different components on a single chip, such as hardware threads, processor cores, memory hierarchy or interconnection networks. The permanent need of evaluating new designs on each of these components motivates the development of tools which simulate the system working as a whole. In this paper, we present the Multi2Sim simulation framework, which models the major components of incoming systems, and is intended to cover the limitations of existing simulators. A set of simulation examples is also included for illustrative purposes.

1 Introduction

The evolution of microprocessors, mainly enabled by technology advances, has led to complex designs that combine multiple physical processing units in a single chip. These designs provide to the operating system (OS) the view of having multiple processors, and thus, different software processes can be scheduled at the same time.

This processor model consists of three major components: the microprocessor core, the cache hierarchy, and the interconnection network. A design improvement on any of these components will result in a performance gain over the whole system. Therefore, current processor architecture trends bring a lot of opportunities for researchers to investigate novel microarchitectural proposals. Below, some design issues on these components are drawn.

Concerning **processor cores**, deep and wide pipelines have been designed, aimed at exploiting the high amount of instruction level parallelism (ILP) present in current workloads. On the other hand, thread level parallelism (TLP) enables to exploit additional sources of independent instructions to increase processor resources utilization. This idea, jointly with an overcome of hardware constraints, resulted in chip multiprocessors (CMPs), which include various cores in a single chip [1].

With respect to **memory hierarchy**, its design is a major concern in current and incoming microprocessors, since long memory latencies act frequently as a performance bottleneck. Current on-chip parallel processing models provide new cache access patterns and offer the possibility of either replicating or sharing caches among processing elements. This fact rises the need to evaluate tradeoffs between memory hierarchy configuration and processor cores/threads structure.

Finally, **interconnection networks** (or interconnects) serve as communication medium for processor components (mainly processor cores). In an environment where caches from different processors share memory blocks, the interconnect is in charge of transmitting coherence messages generated by the cache controllers. Research in this field tries to increase network performance by focusing on new topologies, switching and flow control mechanisms, routing algorithms or fault tolerance techniques.

In this paper we present Multi2Sim, which integrates processor cores, memory hierarchy and interconnection network in a tool that enables their evaluation. The rest of this paper is organized as follows. Section 2 presents an overview of existing processor simulators. Section 3 describes the Multi2Sim structure, while Section 4 discusses the integrated features to support multithreading and multicore simulation. Examples including simulation results are shown in Section 5. Finally, Section 6 presents some concluding remarks.

2 Related Work

Multiple simulation environments aimed at evaluating computer architecture proposals have been developed. The most widely used simulator in recent years has been SimpleScalar [2], which models an out-of-order superscalar processor. Lots of extensions have been applied to Sim-

pleScalar to model in a more accurate manner certain aspects of superscalar processors. For example, the HotLeakage simulator [3] quantifies leakage energy consumption.

SimpleScalar is quite difficult to extend to model new parallel microarchitectures without significantly changing its structure. In spite of this fact, various SimpleScalar extensions to support multithreading have been implemented, e.g. SSMT [4], M-Sim [5], or SMTSim [6], but they have the limitation of only executing a set of sequential workloads and implementing a fixed resource sharing strategy among threads.

Multithread and multicore extensions have been also applied to the Turandot simulator [7] [8], which models a PowerPC architecture and has been also used with power measurement aims (PowerTimer [9]). Turandot extensions to parallel microarchitectures are mostly cited (e.g., [10]) but not publicly available.

Both SimpleScalar and Turandot are application-only tools, which directly simulate the behaviour of an application. Such tools have the advantage of isolating the workload execution, so statistics are not affected by the simulation of additional software. The tool proposed in this paper can also be classified as an application-only simulator.

In contrast to the application-only simulators, a set of so-called full-system simulators are available. Such tools are able to boot an unmodified operating system and applications run at the same time over it. Although this model provides higher simulation power, it involves a huge computational load and sometimes unnecessary simulation accuracy.

Simics [11] is an example of generic full-system simulator, commonly used for multiprocessor systems simulation, but unfortunately not freely available. A variety of Simics derived tools has been implemented for specific research purposes in this area. This is the case of GEMS [12], which introduces a timing simulation module to model a complete processor pipeline and a memory hierarchy supporting cache coherence. However, GEMS provides low flexibility of modelling multithreaded designs and it integrates no interconnection network model.

An important feature included in some processor simulators is the *timing-first* approach, provided by GEMS and adopted in Multi2Sim. On such a scheme, a timing module traces the state of the processor pipeline while instructions traverse it, possibly in a speculative state. Then, a functional module is called to actually execute the instructions, so the correct execution paths are always guaranteed by a previously developed robust simulator. The *timing-first* approach confers efficiency, robustness, and the possibility of performing simulations on different levels of detail. Our proposal adopts the *timing-first* simulation with a functional support that, unlike GEMS, need not simulate a whole operating system, but is still capable of executing parallel workloads, with dynamic threads creation.

The last cited simulator is M5 [13], which provides support for out-of-order SMT-capable CPUs, multiprocessors and cache coherency, and runs in both full-system and application-only modes. The limitations lie once again in the low flexibility of multithreaded pipeline designs.

3 Basic simulator description

Multi2Sim [14] has been developed integrating some significant characteristics of popular simulators, such as separate functional and timing simulation, SMT and multiprocessor support and cache coherence. Multi2Sim is an application-only tool intended to simulate final MIPS32 executable files. With a MIPS32 cross-compiler (or a MIPS32 machine) one can compile his own program sources, and test them under Multi2Sim. This section deals with the process of starting and running an application in a cross-platform environment, and describes briefly the three implemented simulation techniques (functional, detailed and event-driven simulation).

3.1 Program Loading

Program loading is the process in which an executable file is mapped into different virtual memory regions of a new software context, and its register file and stack are initialized to start execution. In a real machine, the operating system is in charge of these actions, but an application-only tool should manage program loading during its initialization.

Executable File Loading. The executable files output by *gcc* follow the ELF (Executable and Linkable Format) specification. An ELF file is made up of a header and a set of sections. Some Linux distributions include the library `libbfd`, which provides types and functions to list the sections of an ELF file and track their main attributes (starting address, size, flags and content). When the flags of an ELF section indicate that it is *loadable*, its contents are copied into memory after the corresponding starting address.

Program Stack. The next step of the program loading process is to initialize the process stack. The aim of the program stack is to store function local variables and parameters. During the program execution, the stack pointer (*$sp* register) is managed by the own program code. However, when the program starts, it expects some data in it, namely the program arguments and environment variables, which must be placed by the program loader.

Register File. The last step is the register file initialization. This includes the *$sp* register, which has been progressively updated during the stack initialization, and the

PC and *NPC* registers. The initial value of the *PC* register is specified in the ELF header of the executable file as the program entry point. The *NPC* register is not explicitly defined in the MIPS32 architecture, but it is used internally by the simulator to handle the branch delay slot.

3.2 Simulation Model

Multi2Sim uses three different simulation models, embodied in different modules: a functional simulation engine, a detailed simulator and an event-driven module —the latter two perform the timing simulation. To describe them, the term *context* will be used hereafter to denote a software entity, defined by the status of a virtual memory image and a logical register file. In contrast, the term *thread* will refer to a processor hardware entity comprising a physical register file, a set of physical memory pages, a set of entries in the pipeline queues, etc. The three main simulation techniques are described next.

Functional Simulation, also called *simulator kernel*. It is built as an autonomous library and provides an interface to the rest of the simulator. This engine does not know of hardware threads, and owns functions to create/destroy software contexts, perform program loading, enumerate existing contexts, consult their status, execute machine instructions and handle speculative execution. The supported machine instructions follow the MIPS32 specification [15] [16]. This choice was basically motivated by a fixed instruction size and formats, which enable a simple instruction decoding.

An important feature of the simulation kernel, inherited from SimpleScalar [2], is the checkpointing capability of the implemented memory module and register file, thinking of an external module that needs to implement speculative execution. In this sense, when a wrong execution path starts, both the register file and memory status are saved, reloading them on the misprediction detection.

Detailed Simulation. The Multi2Sim detailed simulator uses the functional engine to perform a *timing-first* [12] simulation: in each cycle, a sequence of calls to the kernel updates the state of existing contexts. The detailed simulator analyzes the nature of the recently executed machine instructions and accounts the operation latencies incurred by hardware structures.

The main simulated hardware consists of pipeline structures (stage resources, instruction queue, load-store queue, reorder buffer...), branch predictor (modelling a combined *bimodal-gshare* predictor), cache memories (with variable size, associativity and replacement policy), memory management unit, and segmented functional units of configurable latency.

Event-Driven Simulation. In a scheme where functional and detailed simulation are independent, the implementation of the machine instructions behaviour can be centralized in a single file (functional simulation), increasing the simulator modularity. In this sense, function calls that activate hardware components (detailed simulation) have an interface that returns the latency required to complete their access.

Nevertheless, this latency is not a deterministic value in some situations, so it cannot be calculated when the function call is performed. Instead, it must be simulated cycle by cycle. This is the case of interconnects and caches, where an access can result in a message transfer, whose delay cannot be computed *a priori*, justifying the need of an independent event-driven simulation engine.

4 Support for Multithreaded and Multicore Architectures

This section describes the basic simulator features that provide support for multithreaded and multicore processor modelling. They can be classified in two main groups: those that affect the functional simulation engine (enabling the execution of parallel workloads) and those which involve the detailed simulation module (enabling pipelines with various hardware threads on the one hand, and systems with several cores on the other).

4.1 Functional simulation: parallel workloads support

The functional engine has been extended to support parallel workloads execution. In this context, parallel workloads can be seen as tasks that dynamically create child processes at runtime, carrying out communication and synchronization operations. The supported parallel programming model is the one specified by the widely used POSIX Threads library (`pthread`) shared memory model [17].

In a multithreaded environment, some studies suggest using a set of sequential workloads [18]. The reason is that multiple resources are shared among hardware threads, and processor throughput can be evaluated more accurately when no contention appears due to communication between contexts. In contrast, multicore processor pipelines are fully replicated, and an important contention point is the interconnection network. The execution of multiple sequential workloads exhibits only some interconnect activity in form of L2-L1 cache transfers, but no coherence actions can occur between processes having disjoint memory maps. Thus, in order to evaluate multicore processors, it makes sense to support and run parallel workloads with shared memory locations, whose distributed access can stress the interconnection network.

Actual parallel workloads require special hardware support (machine instructions), as well as low level software support (system calls) that enable threads spawning, synchronization and termination. Each of these issues are described below, jointly with a brief description of the POSIX threads management:

Instruction set support. When the processor hardware supports concurrent threads execution, the parallel programming requirement that directly affects its architecture is the existence of critical sections, which cannot be executed simultaneously by more than one thread. CMPs or multithreaded processors must stall the activity of a hardware thread when it tries to enter a critical section occupied by other thread.

The MIPS32 approach implements the mutual exclusion mechanism by means of two machine instructions (`LL` and `SC`), defining the concept of RMW (read-modify-write) sequence [16]. An RMW sequence is a set of instructions, embraced by a pair `LL`-`SC` that run atomically on a multiprocessor system. The cited machine instructions do not enforce an RMW sequence, but the output value of `SC` informs of the RMW success or failure.

Operating system support. Tracing the execution of a parallel workload, the operating system support required by `pthread` is formed of system calls i) to spawn/destroy a thread (`clone`, `exit_group`), ii) to wait for child threads (`waitpid`), iii) to communicate and synchronize threads with system pipes (`pipe`, `read`, `write`, `poll`) and iv) to wake up suspended threads using system signals (`sigaction`, `sigprocmask`, `sigsuspend`, `kill`).

POSIX Threads parallelism management. Applications programmed with `pthread` can be simulated without changes using Multi2Sim. This library introduces user code which handles parallelism by means of the described subset of machine instructions and system calls. However, the fact of having thread management code mingled with application code must be taken into account, as it constitutes a certain overhead which could affect final results. Further details on this consideration can be found in [14].

4.2 Detailed simulation: Multithreading support

Multi2Sim supports a set of parameters that specify how stages are organized in a multithreaded design. Stages can be shared among threads or private per thread [19] (except the *execute* stage, which is shared by definition of multithread). Moreover, when a stage is shared, there must be an algorithm which schedules a thread every cycle on the stage. The modelled pipeline is divided into five stages, described below.

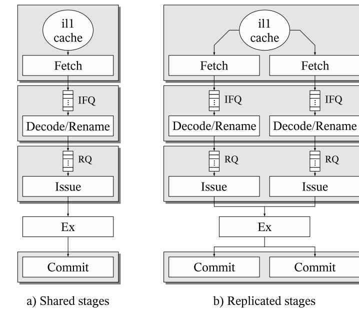

Figure 1. Examples of pipeline organizations

The *fetch* stage takes instructions from the L1 instruction cache and places them into an IFQ (instruction fetch queue). The *decode/rename* stage takes instructions from an IFQ, decodes them, renames their registers and assigns them a slot in the ROB (reorder buffer) and IQ (instruction queue). Then, the *issue* stage consumes instructions from the IQ and sends them to the corresponding functional unit. During the *execution* stage, the functional units operate and write their results back into the register file. Finally, the *commit* stage retires instructions from the ROB in program order. This architecture is analogous to the one modelled by the SimpleScalar tool set [2], but uses a ROB, an IQ (instruction queue) and a physical register file, instead of the RUU (register update unit).

Figure 1 illustrates two possible pipeline organizations. In a) all stages are shared among threads, while in b) all stages (except *execute*) are replicated as many times as supported hardware threads. Multi2Sim allows to evaluate different stage sharing strategies, as well as different algorithms that schedule stage resources in each cycle. Depending on the stages sharing and thread selection policies, a multithread processor can be classified as fine-grain (FGMT), coarse-grain (CGMT) or simultaneous multithread (SMT).

A FGMT processor switches threads on a fixed schedule, typically on every processor cycle. In contrast, a CGMT processor is characterized by a thread switch induced by a long latency operation or a thread quantum expiration. Finally, an SMT processor enhances the previous ones with a more aggressive instruction issue policy, which is able to issue instructions from different threads in a single cycle. The simulator parameters that specify the sharing strategy of pipeline stages among threads, and thus the kind of multithreading, are summarized in Table 1. Again, [14] gives a detailed description of all possible values these parameters may take.

Table 1. Combination of parameters for different multithread configurations

	FGMT	CGMT	SMT
`fetch_kind`	`timeslice`	`switchonevent`	`timeslice/ multiple`
`fetch_priority`	-	-	`equal/icount`
`decode_kind`	`shared/ timeslice/ replicated`	`shared/ timeslice`	`shared/ timeslice/ replicated`
`issue_kind`	`timeslice`	`shared/ timeslice`	`replicated`
`retire_kind`	`timeslice`	`timeslice`	`timeslice/ replicated`

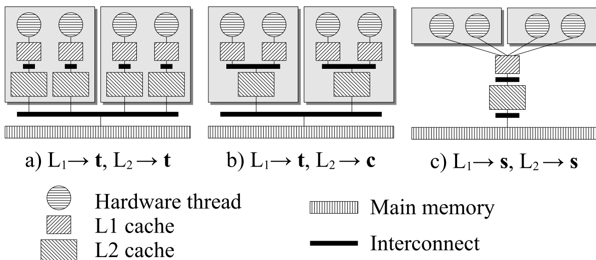

Figure 2. Evaluated cache distribution designs

4.3 Detailed simulation: Multicore support

A multicore simulation environment is basically achieved by replicating the data structures that represent a single processor core. The zone of shared resources in a multicore processor starts with the memory hierarchy. When caches are shared among cores, some contention can exist when they are accessed simultaneously. In contrast, when they are private per core, a coherence protocol (e.g. MOESI [20]) is implemented to guarantee memory consistency. Multi2Sim implements in its current version a split-transaction bus as interconnection network, extensible to any other topology of on-chip networks.

The number of interconnects and their location vary depending on the sharing strategy of data and instruction caches. Figure 2 shows three possible schemes of sharing L1 and L2 caches (t = private per thread, c = private per core, s = shared), and the resulting interconnects for a dual-core dual-thread processor.

5 Results

This section presents some simulation experiments using Multi2Sim, illustrating the simulator application on one hand, and checking its correctness on the other. These experiments i) test different multithread pipeline configurations, ii) explore different bus widths and iii) trace the network traffic executing a parallel workload. In all cases, the simulated machine includes 64KB separate L1 instruction and data caches, 1MB unified and shared among threads L2 cache, private physical register files of 128 entries, and fetch, decode, issue and commit width of 8 instructions per cycle.

i) Multithread Pipeline Organizations. Figure 3 shows the results for four different multithreaded implementations: FGMT, CGMT, SMT with equal thread priorities and SMT with ICOUNT (giving priority to those threads with less instructions in the pipeline [21]). Figure 3a shows the average number of instructions issued per cycle, while Figure 3b represents the global IPC (i.e., the sum of the IPCs achieved by the different threads), executing benchmark *176.gcc* from the SPEC2000 suite with one instance per hardware thread, and varying the number of threads.

Results are in accordance with the ones published by Tullsen et al [18], where CGMT and FGMT processors performs slightly better when the number of threads increases up to four threads. Besides, an SMT processor shows not only higher performance for any number of threads, but also higher scalability, both with equal and variable thread priorities.

ii) Bus Width Evaluation. This experiment shows how the bus width impacts on processor performance, resulting in different number of contention cycles during data transfers. For this test, we assume MOESI requests of 8 bytes and cache blocks of 64 bytes, so network messages can have either 8 bytes (only a MOESI request) or 72 bytes (MOESI request + block data). The executed workload is *fft*, which belongs to the SPLASH2 suite, a set of parallel benchmarks.

Figure 4 represents the average contention cycles per transfer. Because no message larger than 72 bytes will be transferred, at least a 72-byte bus width is required to send any message in a single bus cycle and minimize contention. However, results show that a bus width more than three times smaller provides (for this workload) almost the same benefits.

iii) Interconnect Traffic Evaluation. This experiment shows the activity of the interconnection network during the execution of the *fft* benchmark with the same processor configuration described above, for a 16-byte bus width. Figure 5a represents the fraction of total bus bandwidth used in the network connecting the L1 caches and the common L2 cache, taking intervals of 10^4 cycles. Figure 5b represents the same metric referring to the interconnect between L2 and main memory (MM). As one can see, traffic distribution is quite irregular, showing some peaks of interconnect activity at some execution intervals.

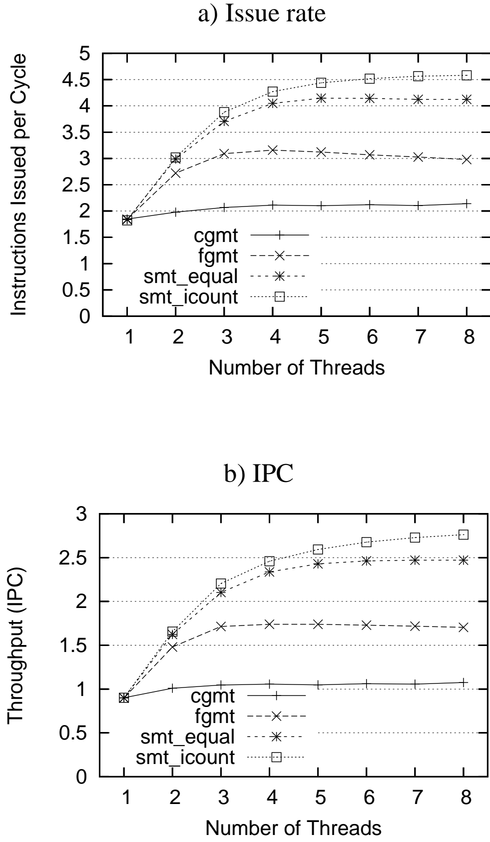

Figure 3. Issue rate and IPC with different multithreaded designs

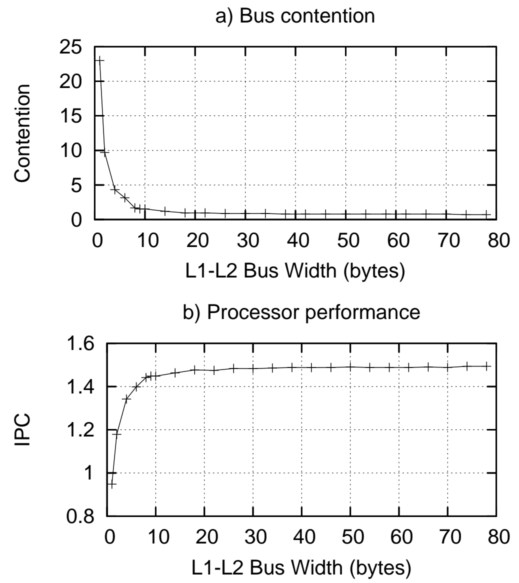

Figure 4. Performance for different bus widths simulating *fft*

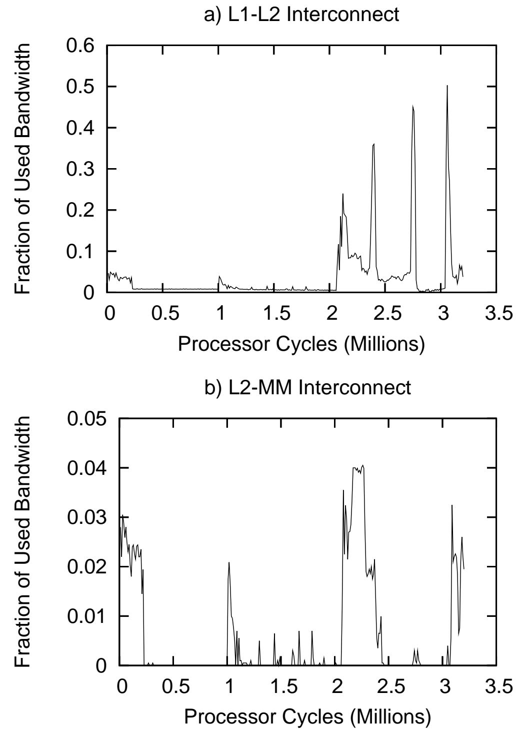

Figure 5. Traffic distribution in L1-L2 and L2-MM interconnects

6 Conclusions

In this paper, we presented Multi2Sim, a simulation framework that integrates important features of existing simulators and extends them to provide additional functionality. Regarding the features adopted from other tools, we can cite the basic pipeline architecture (SimpleScalar), the timing first simulation (Simics-GEMS) or the support to cache coherence protocols.

Among the extensions of Multi2Sim, we find the simulation of sharing strategies of pipeline stages, memory hierarchy configurations, multicore-multithread combinations and an integrated interface with the on-chip interconnection network. These features make Multi2Sim suitable for the evaluation of state-of-the-art processors, covering hot topics in the computer architecture field. In this paper, we showed some guidance examples on how to use these simulator characteristics.

The source code of Multi2Sim is written in C and can be downloaded at [14].

Acknowledgements

This work was supported by CICYT under Grant TIN2006-15516-C04-01, by Consolider-Ingenio 2010 under Grant CSD2006-00046 and by the Generalitat Valenciana under grant GV06/326.

References

[1] AMD Athlon™ 64 X2 Dual-Core Processor Product Data Sheet. *www.amd.com*, Sept. 2006.

[2] D.C. Burger and T.M. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-1997-1342, 1997.

[3] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan. HotLeakage: A Temperature-Aware Model of Subthreshold and Gate Leakage for Architects. *Univ. of Virginia Dept. of Computer Science Technical Report CS-2003-05*, 2003.

[4] D. Madon, E. Sanchez, and S. Monnier. A Study of a Simultaneous Multithreaded Processor Implementation. In *European Conference on Parallel Processing*, pages 716–726, 1999.

[5] J. Sharkey. M-Sim: A Flexible, Multithreaded Architectural Simulation Environment. *Technical Report CS-TR-05-DP01, Department of Computer Science, State University of New York at Binghamton*, 2005.

[6] D. M. Tullsen. Simulation and Modeling of a Simultaneous Multithreading Processor. *22nd Annual Computer Measurement Group Conference*, December 1996.

[7] M. Moudgill, P. Bose, and J. Moreno. Validation of Turandot, a Fast Processor Model for Microarchitecture Exploration. *IEEE International Performance, Computing, and Communications Conference*, pages 451–457, 1999.

[8] M. Moudgill, J. Wellman, and J. Moreno. Environment for PowerPC Microarchitecture Exploration. *IEEE Micro*, pages 15–25, 1999.

[9] D. Brooks, P. Bose, V. Srinivasan, M. Gschwind, and M. Rosenfield P. Emma. Microarchitecture-Level Power-Performance Analysis: The PowerTimer Approach. *IBM J. Research and Development*, 47(5/6), 2003.

[10] B. Lee and D. Brooks. Effects of Pipeline Complexity on SMT/CMP Power-Performance Efficiency. *Workshop on Complexity Effective Design*, 2005.

[11] 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. *IEEE Computer*, 35(2), 2002.

[12] M. R. Marty, B. Beckmann, L. Yen, A. R. Alameldeen, M. Xu, and K. Moore. GEMS: Multifacet's General Execution-driven Multiprocessor Simulator. *International Symposium on Computer Architecture*, 2006.

[13] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-Oriented Full-System Simulation Using M5. *Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, pages 36–43, Feb. 2003.

[14] www.gap.upv.es/˜raurte/tools/multi2sim.html. R. Ubal Homepage – Tools – Multi2Sim.

[15] MIPS Technologies, Inc. *MIPS32™ Architecture For Programmers*, volume I: Introduction to the MIPS32™ Architecture. 2001.

[16] MIPS Technologies, Inc. *MIPS32™ Architecture For Programmers*, volume II: The MIPS32™ Instruction Set. 2001.

[17] D. R. Butenhof. *Programming with POSIX® Threads*. Addison Wesley Professional, 1997.

[18] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 392–403, June 1995.

[19] J. P. Shen and M. H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. July 2004.

[20] P. Sweazey and A.J. Smith. A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus. *13th Int'l Symp. Computer Architecture*, pages 414–423, June 1986.

[21] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *ISCA*, pages 191–202, 1996.