Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithread Processors

R. Ubal, J. Sahuquillo, S. Petit and P. López
Dept. of Computing Engineering (DISCA)
Universidad Politécnica de Valencia, Valencia, Spain
raurte@gap.upv.es, {jsahuqui, spetit, plopez}@disca.upv.es

Abstract

Current microprocessors are based in complex designs, which are the result of years of investigation, and are supported by new technology advances. The last generation of microprocessors integrates different components on a single chip, such as hardware threads, processor cores, memory hierarchy or interconnection network. There is a permanent need of evaluating new designs on each of these components, and quantifying performance gains on the system working as a whole.

In this technical report, we present the Multi2Sim simulation framework as a model and integration of the main microprocessor components, intended to cover limitations of existing simulators. A set of simulation examples is also included for illustrative purposes.

1 Introduction

The evolution of microprocessors, mainly enabled by new 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, so that different software processes can be scheduled at the same time.

This processing 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 new microarchitectural proposals in order to increase their yield. Below, some design issues on these components are highlighted:

Involving **processor cores**, the current generation of superscalar microprocessors is the result of many efforts of designing deep and wide pipelines that highly exploit instruction level parallelism (ILP). However, the potential of ILP present in current workloads is not high enough to continue increasing hardware units utilization. On the other hand, thread level parallelism (TLP) enables to exploit additional sources of independent instructions to maintain processor resources with higher occupation. This idea, jointly with

Table 1: Multi2Sim's parametrizable options.

Processor Cores	• Cores can be either multithreaded or single threaded, containing in-order or out-of-order pipelines. • Multithreaded cores can implement fine-grain, coarse-grain or simultaneous multithread. • Pipeline resources can be private/shared among threads.
Memory Hierarchy	• L1 and L2 can be shared/private among threads or shared/private among cores. • MOESI snoopy protocol working on multiple cache hierarchy levels.
Interconnection Networks	• A bus as a simple interconnect. • Separated simulator module to implement new interconnects.

an overcome of hardware constraints, resulted in CMPs (chip multiprocessors), which include various cores in a single chip [1]. Each core can integrate either a simple in-order multithreded pipeline [2] or a more complex out-of-order pipeline [3].

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 a new cache access pattern and offer the possibility of either replicating or sharing caches among processing elements. This fact arises the need to evaluate tradeoffs between memory hierarchy configuration and processor cores/threads structure.

Finally, regarding **interconnection networks**, the existence of different caches in the same level of the memory hierarchy sharing memory blocks requires a coherence protocol. This protocol generates messages that must be transferred from one core/thread to another. The transference medium is the interconnection network (or interconnect), which can constitute the bottleneck in the global system performance [4]. In this field, research tries to increase network performance and support links/nodes failures, by proposing new topologies, flow control mechanisms or routing algorithms.

In order to evaluate the impact on the overall performance of any design improvement, it is necessary to model the three major components, as well as their integration in a system working as a whole. In this technical report we present Multi2Sim, which integrates simulation of processor cores, memory hierarchy and interconnection network in a tool that enables their evaluation. Table 1 summarizes the main parametrizable options of Multi2Sim, broken down according to the presented components classification.

The rest of this technical report is structured as follows. Section 2 presents an overview of existing processor simulators and their features. Section 3 describes Multi2Sim with significant development details. Section 4 discusses the added 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 for computer architecture research, have been developed. The most widely used simulator in recent years has been SimpleScalar [5],

which serves as basis of some Multi2Sim modules. It models an out-of-order superscalar processor. Lots of extensions have been applied to SimpleScalar to model in a more accurate manner certain aspects of superscalar processors. For example, the HotLeakage simulator [6] quantifies leakage energy consumption. But SimpleScalar is quite difficult to extend to model new parallel microarchitectures without significantly changing its structure.

In spite of this fact, two SimpleScalar extensions to support multithreading have been implemented in the SSMT [7] and M-Sim [8] simulators. Both tools are useful to implement designs based on simultaneous multithreaded processors, but have the limitation of only executing a set of sequential workloads and the drawback of implementing a fixed resource sharing strategy among threads.

Another approach is the Turandot simulator [9, 10], which models a PowerPC architecture. It has been extended with an SMT and multicore support, and has even been used to power measurement aims. Such extension is provided, for example, by the PowerTimer tool [11]. Turandot extensions to parallel microarchitectures are mostly cited (e.g., [12]) but not publicly available.

Both SimpleScalar and Turandot are application-only tools, that is, simulators that execute directly an application and simulate its interaction with a ficticious underlying operating system (through system calls). Such tools are characterized by not supporting the architecture-specific privileged instruction set, since applications are not allowed to use it. However, they have the advantage of isolating the application execution, so statistics are not affected by a simulation of a real operating system. The proposed tool Multi2Sim can be classified as an application-only simulator, too.

In contrast to the application-only simulators, a set of so-called full-system simulators are available. In such environments, an unmodified operating system is booted over the simulator and applications run at the same time over the simulated operating system. Thus, the entire instruction set is implemented, in conjunction with the interfacing with functional models of many I/O devices, but no emulation of system calls is required. Although this model provides higher simulation power, it involves a huge computational load and sometimes unnecessary simulation accuracy.

Simics [13] 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 created for specific purposes in this research area. This is the case of GEMS [14], which introduces a timing simulation module to model instruction fetch, decode, branch prediction, dynamically instructions schedule and execution and speculative memory hierarchy access. GEMS also specifies a language for implementing cache coherence. However, GEMS provides low flexibility of modelling multithreaded designs and it integrates no interconnection network model. Additionally, any simulator based on Simics must boot and run an operating system, so the high computational load is even increased with each extension.

An important feature of processor simulators is the *timing-first* approach, provided by GEMS and adopted in Multi2Sim. In such a scheme, a timing module traces the state of the processor pipeline while instructions traverse it, possibly in a speculative manner. Then, a functional module is called to actually execute the instructions when they reach the *commit* stage, so the correct execution paths are always guaranteed by a previously

Table 2: Main features of existing simulators.

		SimpleScalar	SSMT	M-Sim	HotLeakage	Turandot	PowerTimer	Simics	GEMS	M5	Multi2sim
Single thread	In-order Pipeline	X	X	X	X				X	X	X
	Out-of-order pipeline	X	X	X	X	X	X		X	X	X
	Power consumption				X		X				
Multithread	Multithread		X	X				X	X	X	X
	FGMT, CGMT, SMT										X
	Resource sharing among threads										X
Multicore	Multicore							X	X	X	X
	Memory hierarchy configuration								X	X	X
	Interconnection network									X	X
	Coherence protocol								X	X	X
Simulation	Application-only	X	X	X	X					X	X
	Full-system							X	X	X	
	Timing-first simulation								X		X

developed robust simulator. The *timing-first* approach confers efficiency, robustness, and the possibility of performing simulations on different levels of detail. The main novelty in this sense is the application of a *timing-first* simulation with a functional support that need not simulate a whole operating system, but is capable to execute parallel workloads, with dynamic threads creation.

The last cited simulator is M5 [15]. This simulator provides support for simple one-CPI functional CPU, out-of-order SMT-capable CPUs, multiprocessors and coherent caches. It integrates the full-system and application-only modes. The limitations lie once again in the low flexibility in multithread pipeline designs.

As summary, Multi2Sim has been developed integrating the most significant characteristics of important simulators, such as separation of functional and timing simulation, SMT and multiprocessor support and cache coherence. Table 2 gathers the main simulator features and marks the differences with existing works. Additional features of Multi2Sim are detailed in further sections.

3 Basic simulator description

This section deals with the main implementation issues that lead to a final simulation environment, and exposes some tips to bring it into use with existing or self-programmed, sequential or parallel workloads. These aspects are addressed by showing some compilation examples, describing briefly the process of loading an ELF executable file into a process virtual memory, and finally analyzing the simulator structure, divided into functional and detailed simulation.

3.1 Simulator and Workloads Compilation

Multi2Sim can be downloaded at [16] as a compressed *tar* file, and has been tested on *i386* and *i86_64* machine architectures, with Linux OS. Once the main file has been downloaded, the following commands should be entered in a command terminal to compile it:

```
tar xzf multi2sim.tar.gz
cd multi2sim
make
```

The simulator compilation requires the library *libbfd*, not preset in some Linux distributions by default. Multi2Sim simulates final executable files, compiled for the MIPS32 architecture, so a cross-compiler is also required to compile your own program sources. This MIPS cross-compiler is usually available as a install package for most Linux distributions. For example, in the case of Suse Linux, the required packages are `cross-mips-gcc-icecream-backend` and `cross-mips-binutils`.

Dynamic linking is not supported, so executables must be compiled statically. A command line to compile a program composed by a single source file named `program.c` could be:

```
mips-linux-gcc program.c -Wall -o program.mips32 -static
```

Executables usually have an approximate minimum size of 4MB, since all libraries are linked with it. For programs that use the math library or `pthread` library, simply include `-lm` or `-lpthread` into the command line.

3.2 Executable File Loader

In a simulation environment, 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. However, Multi2Sim , as other widely used simulators (e.g. SimpleScalar), is not aimed at supporting the simulation of an OS, but only the execution of target applications. For this reason, program loading must be managed by the simulator during the initialization.

The executable files output by *gcc* follow the ELF (Executable and Linkable Format) specification. This format is aimed for shared libraries, core dumps and object code, including executable files. An ELF file is made up of an ELF header, a set of segments and a set of sections. Typically, one or more sections are enclosed in a segment. ELF sections are identified by a name and contain useful data for program loading or debugging. They are labelled with a set of flags that indicate its type and the way they have to be handled during the program loading.

The analysis of the ELF file is provided by the cited `libbfd` library, which embodies the needed functions to list the executable file sections and access their contents. The loader module sweeps all of them and extracts their main attributes: starting address,

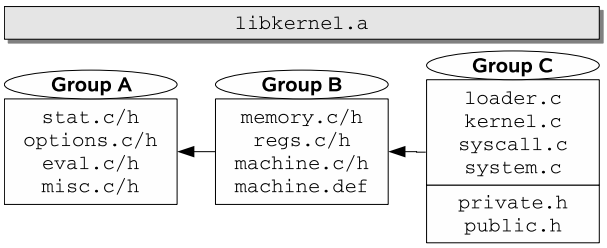

Figure 1: Structure of functional simulation library.

size, flags, and content. When the flags of a section indicate that it is *loadable*, its contents are copied into memory after the corresponding fixed starting address.

The next step of the program loading process is to initialize the process stack. The stack is a memory region with a dynamically variable length, starting at virtual address `0x7fffffff` and growing toward lower memory addresses. The aim of the program stack is to store function local variables and parameters. During the program execution, the stack pointer (register *$sp*) is managed by the own program code. In contrast, when the program starts, it expects some data in it. This fact can be observed looking at the standard header of the *main* function in a C program:

```
int main(int argc, char **argv, char **envp);
```

When the *main* function starts, three parameters are expected starting at the memory location specified by the stack pointer. At address [*$sp*], an integer value represents the number of arguments passed through the command line. At [*$sp*+4], an integer value indicates the memory address corresponding to a sequence of *argc* pointers, which at the same time represent each a null-terminated sequence of characters (program arguments).

Finally, at address [*$sp*+8], another memory address points to an array of strings (i.e. pointers to char sequences). These strings represent the environment variables, accessible through `envp[0]`, `envp[1]`... inside the C program, or by calls to `getenv` functions. Notice that there is no integer value indicating the number of defined environment variables, so the end of the `envp` array is denoted with a final null pointer.

Taking this stack configuration into account, the program loader must write program arguments, environment variables and *main* function arguments into the simulated memory.

The last step is the initialization of the register file. This includes the *$sp* register, which has been progressively updated during the stack initialization, and the *PC* and *NPC* registers. The initial value of register *PC* is specified in the ELF header of the executable file as the program entry point. Register *NPC* is not explicitly defined in the MIPS32 architecture, but it is used internally by the simulator to ease the branch delay slot management.

3.3 Functional Simulation

The functional simulation engine, built as an autonomous library, provides an interface to the rest of the simulator. This engine, also called *simulator kernel*, owns functions to

create/destroy software contexts, perform program loading, enumerate existing contexts, consult their status, execute a new instruction and handle speculative execution.

The supported machine architecture is MIPS32. The main reasons for choosing this instruction set is the availability of an easy to understand architecture specification [17, 18] and the simple and systematic identification of machine instructions, motivated by a fixed instruction size and an instruction decomposition in instruction fields.

As a remark, the difference between the terms *context* and *thread* should be clarified. A *context* is used in this work as a software entity, defined by the status of a virtual memory image and a logical register file. In contrast, a *thread* is used as a processor hardware entity, and can comprise a physical register file, a set of physical memory pages, a set of entries in the pipeline queues, etc. The simulator kernel only handles contexts, and does not know of architecture specific hardware, such as threads or cores.

Figure 1 shows the three different groups of modules that form the kernel. Modules of group A manage statistics, command line options and formula analysis. Group B contains modules to handle virtual memory, registers and instruction execution. The implemented memory module and register file support checkpoints, 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 efficiently saved, reloading them when speculative execution finishes. The file `machine.def` contains the MIPS32 instruction set definition[1].

Finally, modules of group C are described individually below:

- `loader`: program loading, explained above.
- `kernel`: functions to manage contexts. This includes creation and destruction, contexts status query and contexts enumeration. The context status is a combination of flags which describe the current work that a context is doing or able to do. The flags and their meaning are summarized in Table 3.
- `syscall`: implementation of system calls. Since Multi2Sim simulates target applications, the underlying operating system services (such as program loading or system calls) are performed internally by the simulator. This is done by modifying the memory and logical registers status so that the application sees the result of the system call.
- `system`: system event queue, pipes data base and signal handling. The system event queue contains events induced by the simulated operating system, such as contexts wake up due to a write to a pipe, or a timeout. The existence of pending events in the system event queue must be tested periodically. The pipes data base manages creation, destruction and read/write operations to system pipes, regarding that these operations can cause contexts block. At last, the signal handling controls the signals submission among processes, and the execution of the corresponding signal handlers.

[1]The MIPS32 instructions that are not used by the *gcc* compiler are excluded from this implementation. Also instructions belonging to the privileged instruction set are not implemented

Table 3: Flags forming the context status

Value	Meaning
KE_ALIVE	The specified position on the contexts array contains a valid context.
KE_RUNNING	Context is running (not suspended)
KE_SPECMODE	Context is in a wrong execution path; memory and registers are checkpointed.
KE_FINISHED	Context has finished execution with a system call, but is not still destroyed.
KE_EXCL	Context is running instructions in exclusive mode; no other context can run at the same time.
KE_LOCKED	Context is waiting until other context finished execution in exclusive mode.

3.4 Detailed Simulation

The Multi2Sim detailed simulator uses the functional engine contained in `libkernel` to perform an execution-driven simulation: in each cycle, a sequence of calls to the kernel updates the existing contexts state. The detailed simulator analyzes the nature of the recently executed machine instructions and accounts the operation latencies incurred by hardware structures. Each of these structures is implemented in one module of group A or B, as shown in figure 2.

The modules of group A are:

- `bpred.c`: branch predictor. Based on SimpleScalar.
- `cache.c`: caches and TLBs, including model of the MOESI cache coherence protocol.
- `ic.c`: interconnection network. This module follows an event-driven simulation model.
- `mm.c`: memory management unit, whose purpose is to map virtual address spaces of contexts into a single physical memory space. Physical addresses are then used to index caches or branch predictors, without dragging the context identifier across modules.
- `ptrace.c`: pipeline trace dump. In each cycle, the pipeline state can be dumped into a text file, so external programs can process it. A possible application is the graphical representation and navigation across the processor state.

The modules of group B are:

- `mt.c`: definition and management of processor structures, such as reorder buffers, IQs, instruction fetch queues, etc.
- `stages.c`: pipeline stages which define the behaviour of the configured multithreaded processor. The number of stages and their features are specified in section 4.

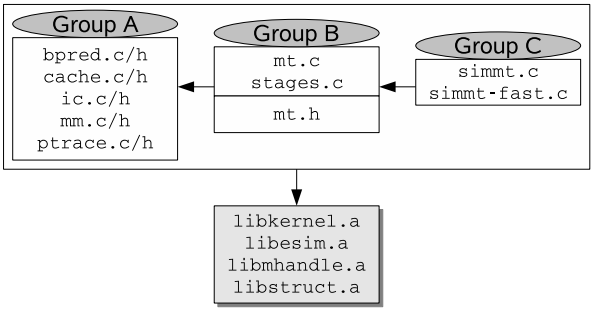

Figure 2: Detailed simulator structure

Finally, modules of group C are:

- `simmt.c`: main program of the Multi2Sim detailed simulator. A wide set of simulation parameters can be varied, and different statistics are shown after the simulation.
- `simmt-fast.c`: main program of the Multi2Sim functional simulator, which uses exclusively the functionality provided by the kernel. No statistics are shown after the simulation.

The simulator makes use of the libraries represented in figure 2, which have been packed in the following files:

- `libkernel.a`: functional simulator kernel.
- `libesim.a`: event-driven simulation library, detailed later.
- `libmhandle.a`: redefinition of `malloc`, `calloc`, `strdup` and `free` to support safe memory management.
- `libstruct.a`: data structures like lists, heaps, hash tables or FIFO buffers.

Motivation of an event-driven simulation library

As explained above, most Multi2Sim modules implement an execution-driven simulation, as SimpleScalar does. This model enables the separation of the functional kernel as an independent library and additionally permits the definition of the instruction set to be located in a single file (`machine.def`). This is an advantage, since the machine instructions behaviour and the flags describing their characteristics are centralized in a simulator module.

In such a model, function calls that activate some processor component (e.g. a cache or predictor) have an interface that receives a set of parameters and returns the latency needed to complete the access. Nevertheless, there are some situations where this latency is not a deterministic value and cannot be obtained in the instant when the function call is performed. Instead, it must be simulated cycle by cycle.

This is the case of interconnects and caches. In a generic topology, the delay of a message transference cannot be determined when the message is injected, because it

depends on the dynamic network state. In addition, this state depends on future message transferences, so it cannot be computed unless advancing the simulation.

Because a cache access in a multithread-multicore environment may cause coherence messages across interconnection networks, the cache access latency cannot be estimated prior to the network access. Thus, the cache module is also implemented with an event-driven model. It provides a mechanism by which cache accesses are labelled. When the execution-driven simulator performs a cache access, periodical calls to the cache module are made to check if an access (identified by its label) has been completed.

4 Support for Multithreaded and Multicore Architectures

This section describes the basic simulator features that provide support for multithreaded and multicore processor modelling. As a first step, the functional simulator engine incorporates the capability of executing parallel workloads, so it is able to guide the detailed simulation of multiple software contexts through the right execution paths. Then, the detailed simulation modules are modified to model different i) pipelines of multithreaded processors, i) memory hierarchy configurations and iii) interconnection networks for multicore processors.

4.1 Functional simulation: parallel workloads support

Following the simulation scheme discussed in Section 3, where a functional simulator is completely independent and provides a clear interface to a set of detailed simulation modules, we extended the functional engine 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 [19].

There is no need to execute parallel tasks to evaluate multithreaded processor designs. In such environments, multiple resources are shared among hardware threads, and processor throughput can be evaluated more accurately when no contention appears due to communication between processes. In other words, evaluation studies should be performed using sequential workloads, as argued in [20]. Nevertheless, the capability of executing parallel workloads confers high flexibility, having also the possibility of executing and evaluating self-developed parallel programs.

In contrast, multicore processor pipelines are fully replicated, and the main contention point is the interconnection network. An execution of multiple sequential workloads does not exhibit any interconnect activity. The reason is that each software context has its own memory map, so physical address spaces of contexts are disjoint from each other. When there are not shared cache blocks among contexts, no coherence action occurs. Thus, it makes sense to execute parallel workloads to evaluate multicore processors, in order to maintain a high interconnect usage caused by coherence protocol transactions.[2]

[2]Notice that these protocols also work when executing sequential workloads, but only a subset of block

When compiling parallel programs and running them over the functional simulator, one can observe the features that a `pthread` application expects from the operating system and the processor architecture to implement. In other words, we can isolate the specific OS interface and the specific subset of machine instructions that provide support to parallel programming. Below, we detail these features and describe how the `pthread` library makes use of them.

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. For single-thread processors, the operating system can handle this issue without any hardware modification, by simply effectuating a context switch when a thread tries to enter an occupied critical section, and maintaining the hardware thread always active. Nevertheless, CMPs or multithreaded processors could have the need to stall the activity of a hardware thread in such situation.

The weakest instruction set requirement to implement mutual exclusion execution is a test-and-set instruction. In this case, when a thread B tries to mark a critical section indicator already set by another thread A, B dives into an active wait loop until the mark is released. In contrast, the MIPS32 approach implements the mutual exclusion mechanism by means of two machine instructions (`LL` and `SC`) and defines the concept of RMW (read-modify-write) sequence [18]. An RMW sequence is a set of instructions, embraced by a pair `LL-SC` that run atomically on a multiprocessor system.

The `LL` instruction loads the contents of a cached memory location into a register and starts an RMW sequence on the current processor. Later, an `SC` instruction stores the contents of this register again into memory and sets its value to 1 or 0 if the RMW sequence completed successfully or not, respectively. As one can observe, the atomicity of the instructions between the `LL-SC` pair is not granted by `LL`, but is ratified or invalidated by `SC`, depending on if the original RMW sequence was or was not interrupted by other processor.

This mechanism (or, alternatively, other consistency enforcing mechanism) must be implemented in a simulation context where functional and detailed simulation are separated, that is, where a functional simulator does not know of processor cores, threads or caches, but only realizes the existence of multiple software contexts. To avoid additional simulator dependencies exclusively induced by these machine instructions, the functional engine executes with explicit atomicity all instructions in the RMW sequence, causing it to succeed in any case. This behaviour is assumed to be far of causing results truncation, since `pthread` RMW sequences are rarely used and are typically formed by only four instructions.

states are reached. For example, the MOESI protocol would never drive blocks to states *owned* or *shared* when no pair of contexts has a common memory map.

Operating system support

The `pthread` library uses a specific subset of system calls in order to implement threads creation, destruction and synchronization, summarized below:

- `clone`, `exit_group`: system calls to spawn a child thread and destroy the current thread, respectively. The parameters of the `clone` system call are the starting address of the code to be executed by the child thread, an optional argument to this function, a pointer to the child thread stack and some flags indicating which software resources will be shared among threads. In the case of `pthreads`, these flags always indicate that parent and child threads will share memory space, file system, file descriptors table and signal handler table. The low order byte of this parameter specifies the signal to be sent to the parent thread when the child terminates.

- `waitpid`: wait for child threads, identifying them by its *pid*.

- `pipe`, `read`, `write`, `poll`: some threads communicate by system pipes. These system calls serve as a way to create, read, write and wait for events over them. Read and write operations over system pipes are also used as threads unidirectional synchronization points: when a read is performed over an empty pipe, the thread is blocked until other thread writes to it.

- `sigaction`, `sigprocmask`, `sigsuspend`, `kill`: these system calls are aimed at managing Linux signals. This mechanism is basically used to wake up suspended threads. `sigaction` establishes a signal handler for a specific signal, that is, the function to be executed when the signal is received. When a thread modifies an entry in the signal handler table, it affects all threads of the same process. `sigprocmask` is used to establish or consult the current signal mask (private per thread). When a thread receives a blocked signal, it is marked as pending, and the corresponding handler will be executed after the signal is unblocked. The system call `sigsuspend` suspends the current thread and sets a temporary signal mask, until a signal not present in this mask is received. At last, `kill` is used to send signals to the thread which corresponds to a specific *pid*.

POSIX Threads parallelism management

The extension of a single-context functional simulator to support dynamic context creation offers two main possibilities to give support to parallel workloads:

- A first approach consists in defining a simulator-specific interface to these programs (by means of system calls) that provides mechanisms to create, synchronize, and destroy threads. As the underlying operating system transactions are implemented in the functional simulator, they do not interfere with the simulation results (a detailed simulator only sees a `syscall` machine instruction that modified certain registers and memory contents).

  In this case, programs should be designed and compiled to be run in this concrete environment, fulfilling the specified parallel programming interface. Although this

restriction could seem too strong, it would be easily accomplished in a parallel environment like the SPLASH2 benchmarks, where all thread handling operations are expressed as C macros. Each macro would be redefined to one or more special system calls intercepted by the simulator.

- Other possibility is to use a standard parallel programming library, such as `pthread`, which allows existing parallel programs to be simulated without changes. In this case, thread management is carried out by the code in the library, which is simulated as it were part of the application code. To perform some privileged operations not feasible with user code (such as thread spawning, suspension, etc.), standard UNIX system calls, previously enumerated, are used by the library. Hence, the functional simulator must implement this set of system calls.

  The flexibility provided by the use of standard libraries has been considered dominant, adopting this approach for the parallelism support implementation. Nevertheless, the fact of having thread management code mingled with application code must be taken into account, in the sense of the overhead it constitutes and how it could affect final results. For this reason, slight comments are included next about how `pthread` deals with threads creation, destruction and synchronization.

Let us suppose that we have the following simple program in C, where the main thread creates a child thread, which prints a message into the screen.

```
void *childfn(void *ptr) {
      printf("child says hello\n");
      return NULL;
}

int main() {
      pthread_t child;
      pthread_create(&child, NULL, childfn, NULL);
      pthread_join(child, NULL);
      return 0;
}
```

In this example, the `pthread` library activation begins during the call to `pthread_create`. When this function is called for the first time, a special thread 1 is created, which is called *manager thread*. This thread is the one that spawns other threads (2, 3, ...) and communicates them. For this aim, a system pipe is created by the main thread 0 before creating the manager thread, through which specific requests will be sent.

The system signals mechanism is used to synchronize threads at the lowest level. `pthread` defines two user signals (`SIGUSR1` and `SIGUSR2`) as a *continue* a *cancel* signal. Usually, when a thread sends a request to the manager, it is suspended (using the `sigsuspend` system call), setting a temporary signal mask that only enables either the *continue* or the *cancel* signal. In the case of `pthread_create`, the created thread 2 is in charge of sending one of these signals (by the `kill` system call) to the suspended creator thread to allow it to continue or to cancel its execution.

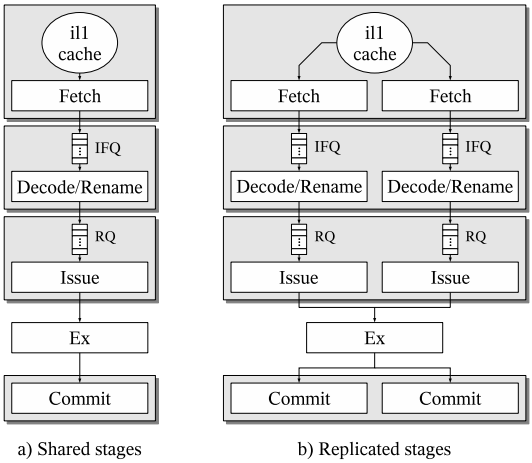

Figure 3: Examples of pipeline designs

Something similar occurs during the call to the `ptrhead_join` function, where thread 0 is suspended, waiting for a *continue* signal. When the child thread 2 reaches the `return` statement, the *continue* signal is sent to thread 0, allowing it to resume its execution. After this, all threads finish with an `exit_group` system call.

These remarks should be considered when simulating programs that make intensive invocations to the library and, thus, spend high portion of its execution time running library thread handling code, system calls or signal handlers. The main overhead of `pthread` is the existence of a manager thread, which must be permanently mapped into a hardware thread in a simulation environment where no software context switches are implemented. For this reason, the program shown in the example requires a simulator configuration with at least c cores and t threads, being $c \times t \geq 3$.

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. Moreover, when a stage is shared, there must be an algorithm which schedules a thread each cycle on the stage. The modelled pipeline is divided into five stages, described below.

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 its registers, assigns them a slot in the ROB (reorder buffer), and places them into a RQ (ready queue) when their input operands become available. Then, the *issue* stage consumes instructions from the RQ and sends them to the corresponding functional unit. During the *execute* 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 [5], but uses a ROB, an IQ (instruction queue) and a physical register file, instead of an integrated RUU (register update unit).

The sharing strategy of each stage can be varied in a multithreaded pipeline [21], with the only exception of the *execute* stage. By definition, multithreading takes advantage of

the sharing of functional units, located in the *execute* stage, increasing their utilization and, consequently, reaching higher global throughput (i.e., IPC). Figure 3 illustrates two possible pipeline designs. In design a) all stages are shared among threads, while in design b) all stages (except *execute*) are replicated as many times as supported hardware threads.

Multi2Sim can be employed to evaluate different stage sharing strategies. Table 4 lists the associated simulator parameters and the possible values they may take. These parameters also specify which policy is used in each stage to process instructions. Depending on the stages sharing and thread selection policies, a multithread design can be classified as fine-grain (FGMT), coarse-grain (CGMT) or simultaneous multithread (SMT). The characteristic parameters for each design are described next.

Simulator Configuration for Different Multithread Architectures

An FGMT processor switches threads on a fixed schedule, typically on every processor cycle. In this case, it makes no sense that stages are private, since only one thread is active at once. Hence, a FGMT design is modelled with all parameters set to `timeslice`, although only a time-shared *fetch* and *issue* stage are strictly necessary [21].

On the other hand, a CGMT processor is characterized by a context switch induced by a long latency operation or a thread quantum expiration. This approach is usually implemented in in-order issue processors, whose execution stage (and hence the pipeline) stalls when a long latency operation is issued. Rather than stall, CGMT allows draining the pipeline after the first instruction following the stalled one, and refilling it with instructions from another thread. Since such stalls may not be detected until late in the pipeline, the drain-refill process incurs a penalty, modelled here as a number of cycles specified by the `switch_penalty` parameter. As only one thread can be active at once in a stage, there is no need to replicate *fetch*, *decode*, *issue* or *commit* logic. Thus, a CGMT simulator configuration is characterized with a `switchonevent` fetch policy and either `shared` or `timeslice` sharing of the rest of the stages.

Finally, SMT designs enhance the previous ones with the instruction issue policy, i.e., with a `shared` issue stage. When only instructions from one thread can be issued in a processor cycle, the so-called vertical waste is palliated, in the sense that any thread with ready instructions can be selected to issue. Therefore, the SMT approach can better exploit the available issue bandwidth by filling empty issue slots with as many instructions as possible, reducing the horizontal waste. All other stages can have an arbitrary sharing, depending on hardware complexity constraints or throughput requirements, although most common designs use `timeslice` at fetch, decode and commit stages.

Table 5 summarizes the combinations of parameter values that model the described multithread architectures.

In an SMT design, it is also possible to replicate the *fetch* stage to allow instructions from different threads to enter the pipeline in a single cycle. This choice can be implemented without a considerable amount of extra hardware by means of instruction cache subbanking. Multi2Sim supports it by changing the `fetch_kind` parameter to `multiple` fetch. In this scenario, more advanced designs define mechanisms to prioritize threads. As researches show [22], global throughput is increased by giving fetch precedence to threads which are not likely to stall the pipeline when executing long latency operations.

Table 4: Simulator parameters to vary pipeline design.

Value	Meaning
	Parameter mt_fetch_kind
`timeslice`	The fetch stage is shared among hardware threads. Each cycle, `fetch_width` instructions are fetched from one different thread in a round-robin fashion.
`switchonevent`	Fetch stage shared. A single thread is continuously fetched while no thread switch causing event occurs.
`multiple`	Fetch stage is shared. `fetch_width` is given out either fairly to all running threads or applying thread priority policies (`mt_fetch_priority` parameter).
	Parameter mt_decode_kind
`shared`	The previous stage (fetch) must deliver instructions to a single IFQ. Each cycle, `decode_width` instructions are decoded and renamed.
`timeslice`	In the previous stage, each thread must deliver instructions to a dedicated IFQ. The decode and rename logic are shared, and act each cycle over `decode_width` instructions from a single IFQ in a round-robin fashion.
`replicated`	The decode stage contains multiple IFQs and replicated decode and rename logic. Each cycle, `decode_width` instruction from all hardware threads are processed.
	Parameter mt_issue_kind
`shared`	When instructions are ready to be issued, a single ready queue (RQ) is used for all threads. Each cycle, `issue_width` instructions are scheduled from this RQ to the corresponding functional units.
`timeslice`	The issue stage contains one RQ per thread, but shared issue logic, dedicated in each cycle to a specific thread in a round-robin fashion.
`replicated`	Multiple instructions are taken from multiple RQ (one per thread) in each cycle, with a maximum of `issue_width` instructions. This option is typical of a SMT implementation.
	Parameter mt_commit_kind
`timeslice`	The commit stage takes `commit_width` instructions from the ROB of one thread each cycle in a round-robin fashion.
`replicated`	The commit logic is replicated for each ROB, and `commit_width` instructions are committed each cycle from each thread.

Table 5: 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`
`commit_kind`	`timeslice`	`timeslice`	`timeslice/ replicated`

Setting the simulator parameter `fetch_priority` to `equal`, the fetch width is given out fairly among all threads, while a value of `icount` in the same parameter indicates that thread priorities change dynamically using the ICOUNT policy. With this policy, higher priority is assigned to those threads with a lower number of instructions in the pipeline queues (IQ, IFQ, RQ, etc).

4.3 Detailed simulation: Multicore support

A multicore simulation environment is basically achieved by replicating all data structures that represent a single processor core. In a single cycle, the function calls that implement pipeline stages are also replicated for each core.

The zone of shared resources in a multicore processor starts with the memory hierarchy (caches or main memory). When caches are shared among cores, some contention can exist when they try to be accessed simultaneously. In contrast, when they are private per core, a coherence protocol (in this case MOESI) is implemented to guarantee memory consistency. This protocol generates coherence messages and cache block transferences that require a communication medium, referred to as interconnection network.

Interconnects constitute the main bottleneck in current multiprocessor systems, so their design and evaluation is an important feature in research oriented simulation tools. Particularly, Multi2Sim implements (in its basic version) a simple bus, extensible to any other topology of current on-chip networks (OCNs) in multicore processors. Additionally, the number of interconnects and their location vary depending on the sharing strategy of data and instruction caches.

Figure 4 shows possible schemes of sharing L1 and L2 caches (t means that the cache is private per thread, c means private per core, and s means shared for the whole CMP). In all combinations, a dual core dual thread processor is represented. For instance, Figure 4b represents L1 caches private per thread, and L2 caches shared among threads. With this concrete configuration, three interconnects are needed: one that binds two L1 caches with an L2 cache in core 1, another with the same utility in core 2, and the last one that binds the L2 caches with main memory.

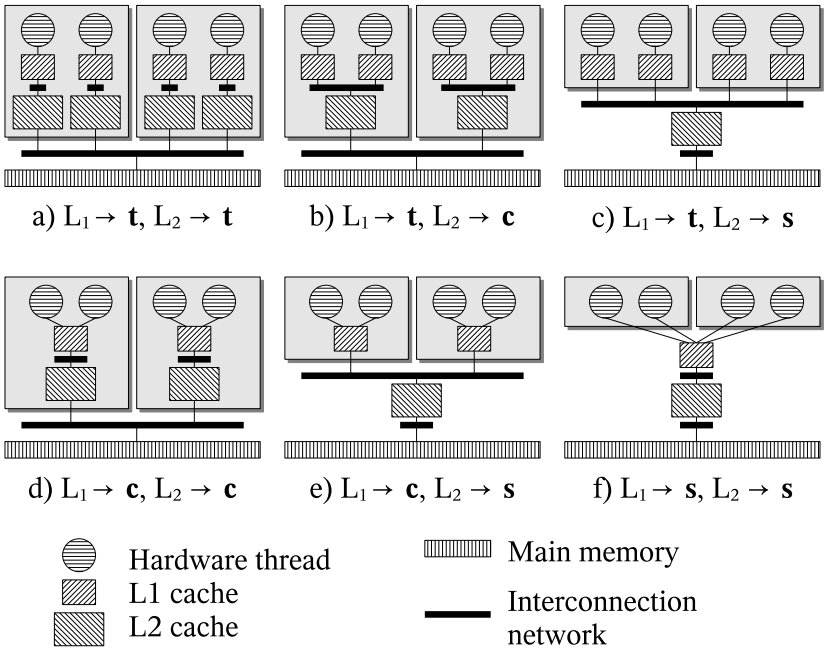

Figure 4: Evaluated cache distribution designs

5 Results

This section shows simulation experiments using Multi2Sim. Rather than testing the processor behaviour with specific designs, the goal of these experiments is to illustrate some simulator application on one hand, and to check the correctness of the simulator on the other. The experiments can be classified into two main groups: multithread pipeline designs and multicore processors/interconnects.

5.1 Multithread Pipeline Designs

Figure 5 shows the results for four different multithreaded implementations: FGMT, CGMT, SMT with equal priorities and SMT with ICOUNT. 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. Figure 5a shows the average number of instructions issued per cycle, while figure 5b 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 published by Tullsen et al [20]. A CGMT processor performs slightly better when the number of threads is increased up to four threads. The reason is that four threads are enough to guarantee that there will be always (or almost always) some non-stalled thread that can replace the current one when it stalls on a long latency operation.

An FGMT processor behaves similarly. Again, four threads are enough to always

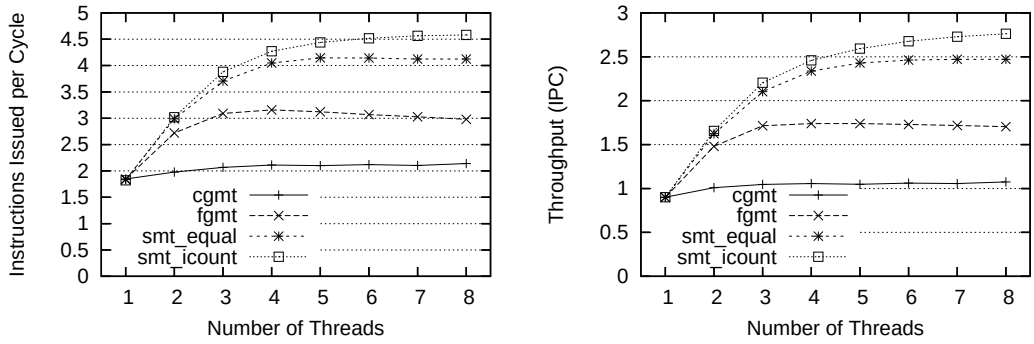

Figure 5: a) Issue rate and b) IPC with different multithreaded designs

provide an available thread with ready instructions in each cycle. The improvement over CGMT is basically due to the inexistence of context switch and its subsequent penalty.

At last, SMT shows not only higher performance for any number of threads, but also higher scalability, both with equal and variable thread priorities. While CGMT and FGMT reach highest performance with four threads (by filling vertical waste), SMT continues improving performance with a higher number of threads, by filling empty issue slots in a single cycle with ready instructions from other threads (horizonal waste).

5.2 Multicore processors/interconnects

Three different experiments illustrate how the simulator works modelling a multicore processor. They evaluate i) the MOESI protocol under a specific memory hierarchy configuration, ii) the distribution of the contention cycles for different bus widths, and iii) the intensity of data traffic through the interconnection network for a given workload.

5.2.1 Evaluation of the MOESI Protocol

To validate the correctness of the MOESI protocol implementation, two experiments were performed: the first one simulates a replicated sequential workload (*gcc* from the SPEC2000 suite), while the second one executes a parallel workload (*fft* from the SPLASH2 suite) spawning two contexts (one manager and one child context). Both experiments model a 4-core processor with one hardware thread per core, 64KB private and separate L1 instruction and data caches and 1MB unified and shared among cores L2 cache.

In a scheme where a set of processing nodes with private L1 caches share a L2 cache, the coherence protocol must guarantee memory consistency among cache blocks. In the case of the MOESI protocol, this is manifested in a set of MOESI requests and cache blocks transferences across the network interconnecting the L1 caches and the common L2 cache. When the nodes access a L1 cache block, they may change the associated MOESI state.

Multi2Sim provides a set of per cache statistics that indicate the percentage of blocks in any MOESI state, averaging all simulation cycles. Figure 6 shows the results for the selected workloads. The represented block states distribution corresponds to the private

L1 cache of the hardware thread 0, that is, the first of the replicated contexts for the *gcc* benchmark, and the main context of the *fft* workload.

The key difference between both distributions is the inexistence of blocks in a *shared* or *owned* state when sequential workloads are executed (Figure 6a). The reason is that these states can be reached only when the same memory block is allocated into multiple caches or when a cache contains the unique valid copy of a shared block, respectively. None of these situations occur when no physical address space is shared among contexts.

Additionally, there is a higher occurrence of the *invalid* state in the parallel workload execution (Figure 6b). The reason is that, in this case, blocks are continuously invalidated due to MOESI actions. In contrast, a sequential workload execution only exhibits invalid blocks during the initialization process, when the caches do not contain valid data yet.

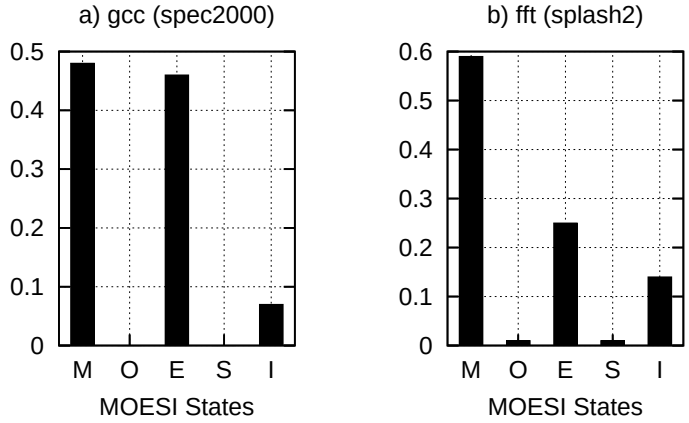

Figure 6: Fraction of blocks with each MOESI states

5.2.2 Bus Width Evaluation

The second experiment concerning multicore processors shows how the interconnect bus width affects processor performance. Multi2Sim calculates and outputs the cumulative number of contention cycles that each transference involves. Contention cycles appear when a node tries to send a message through the interconnect, but finds it busy transferring previous pending messages.

In this experiment, two values must be fixed: the MOESI request size and the cache block size. A MOESI request is composed of a block address and a code indicating the MOESI action, so 8 bytes can conform a reasonable size. On the other hand, the cache block size is assumed to have a length of 64 bytes.

Figure 7 represents the average contention cycles per transference, dividing the accumulative number of contention cycles by the number of transferences. We can distinguish two kinds of MOESI transactions: some including only a MOESI request, and some including additional data to update cache contents. Thus, transferred messages will have either 8 or 72 bytes, so a bus width of 72 bytes would provide the lowest contention. However, the results show that a bus width more than three times smaller provides (for this workload) almost the same benefits. This observation could be useful, for instance, to explore tradeoffs between bus cost and processor performance.

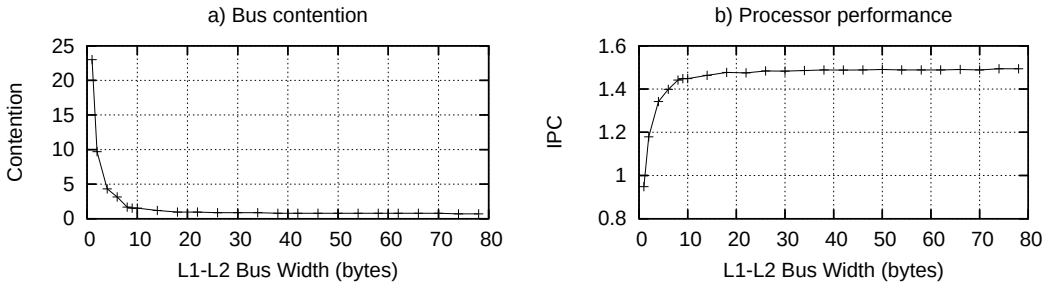

Figure 7: Performance for different bus widths simulating *fft*

5.2.3 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, and a bus width of 16 bytes. Figure 8a 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 8b represents the same metric referring to the interconnect between L2 and main memory (MM). This kind of plots permits to analyze how actions performed to tackle coherence and consistency are spread across the time. The representation of traffic distribution may help, for example, to evaluate a new coherence protocol, or to inspect the behaviour of a parallel application.

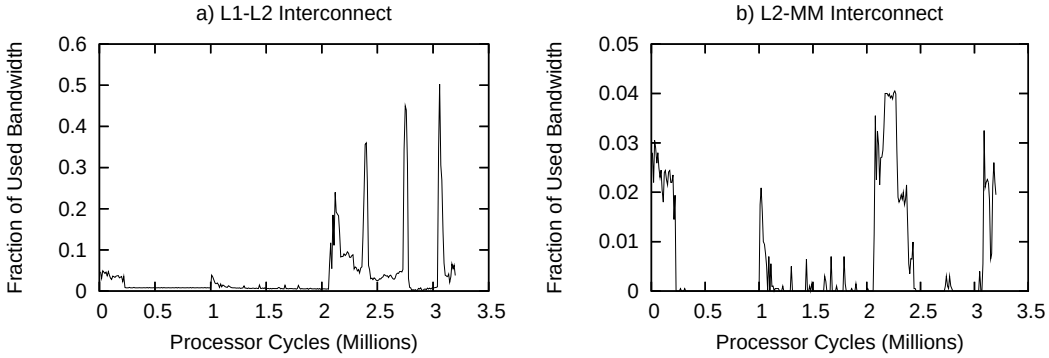

Figure 8: Traffic distribution in L1-L2 and L2-MM interconnects

6 Conclusions

In this technical report, we presented Multi2Sim, a simulation framework that integrates some features of existing simulators and extends them to provide additional functionality.

Among the adopted features, we can cite the basic pipeline architecture (SimpleScalar), the timing first simulation (Simics-GEMS) or the support to cache coherence protocols. On the other hand, some of the own Multi2Sim extensions are the sharing strategies of pipeline stages, memory hierarchy configurations, multicore-multithread combinations

and an integrated interface with the on-chip interconnection network. We have shown some guidance examples on how to use some simulator features.

As this tool has mainly research aims, it has been built to serve as support for future works, such as development and evaluation of performance improvement techniques. Multi2Sim is foreseen to be used both in the field of computer architecture and interconnection networks. The source code of Multi2Sim, written in C, can be downloaded at [16].

Acknowledgements

This work was supported by CICYT under Grant TIN2006-15516-C04-01 and by CONSOLIDER-INGENIO 2010 under Grant CSD2006-00046.

References

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

[2] Cameron McNairy and Rohit Bhatia. Montecito: A Dual-Core, Dual-Thread Itanium Processor. *IEEE Micro*, 25(2), 2005.

[3] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 chip: a dual-core multithreaded processor. *IEEE Micro*, 24(2), 2004.

[4] T. Pinkston and J. Duato. Multicore and Multiprocessor Interconnection Networks. *Acaces 2006, L'Aquila (Italy)*, 2006.

[5] D.C. Burger and T.M. Austin. The SimpleScalar Tool Set, Version 2.0. *Computer Architecture News*, 25(3), 1997.

[6] 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.

[7] Dominik Madon, Eduardo Sanchez, and Stefan Monnier. A Study of a Simultaneous Multithreaded Processor Implementation. In *European Conference on Parallel Processing*, pages 716–726, 1999.

[8] 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.

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

[10] M. Moudgill, J. Wellman, and J. Moreno. Environment for PowerPC Microarchitecture Exploration. *IEEE Micro*, 1999.

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

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

[13] 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), 2002.

[14] 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.

[15] 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)*, Feb. 2003.

[16] www.gap.upv.es/~raurte/multi2sim.html. R. Ubal Homepage – Tools – Multi2Sim.

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

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

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

[20] 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*, June 1995.

[21] John P. Shen and Mikko H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. July 2004.

[22] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *ISCA*, pages 191–202, 1996.