Retargetable Cache Simulation Using High Level Processor Models*

Rajiv Ravindran Rajat Moona
Department of Computer Science & Engineering
Indian Institute of Technology
Kanpur, U.P., 208016, India
{rajiva,moona}@cse.iitk.ac.in

Abstract

During processor design, it is often necessary to evaluate multiple cache configurations. This paper describes the design and implementation of a retargetable on-line cache simulator. The cache simulator has been implemented using a retargetable instruction set simulator from the Sim-nML [9] processor description language. The retargetability helps in cache simulation and evaluation much before the actual processor design.

1. Introduction

During the design of modern embedded systems, it is necessary to automatically generate processor and application development tools like assemblers, disassemblers, compilers, instruction simulators etc. Automated generation of such tools yields faster turn-around time with lower costs for the system design and simplify the process of design changes. Most such tools are system and processor specific. However, with ever increasing complexity of systems and special purpose processors, there is a strong need for generic and modular generator tools that could automatically generate processor and application specific tools from a high level description of the processor or system. Such generator tools replace the system or processor specific tools and provide a generic integrated environment for processor development. For a designer of the system, these generator tools are useful as they allow him to explore several alternatives early in the design phase. We have developed one such environment. In our environment, the processors are modeled at a very high level of abstraction in *Sim-nML* specification language [9].

We use *Sim-nML* to describe the instruction set architecture of the processor from which various tools to aid processor design are generated automatically. As part of the integrated development environment, we have developed a cache simulation environment. This provides a mechanism to simulate various caching policies. The designer can use this cache simulator to study the trade-offs between different caching policies by varying cache specific parameters.

During the design of a processor, it is essential to characterize the data and address access patterns of the given application. This helps the designer in choosing an optimal caching policy and maximize the performance of the given application on the processor. In an embedded system design scenario, the application design can be done even before the actual design of the processor. It also helps in evaluation of the application program for timing constraints. The other alternatives to this are to either design the processor or develop a processor specific simulator. Both approaches are clearly not suited in the embedded processor design world where the designer would want to iterate over multiple design options and decide on the best for the given application domain in a short period of time. The cache simulator is a step towards our original goal of complete system simulation consisting of a core microprocessor simulator, a cache simulator, peripherals etc. in a retargetable manner from *Sim-nML*.

We provide an on-line cache simulation using a retargetable application specific instruction set simulator. On-line cache simulator keeps track of the instruction and data addresses depending on the caching policy at run time. This involves running the application on a functional simulator for the processor and tracing the instruction and data memory references. The cache simulator is built upon the functional simulator - *Fsim* [1]. A retargetable functional simulator generator - *Fsimg*, generates a processor specific functional simulator using the processor model written in *Sim-nML*. As the functional simulator simulates the execution of the given program, on-line cache simulation is performed using specific *canonical functions* (refer section 5). The instruction or data addresses are passed as parameters to the cache simulator, which simulate the caching behavior by keeping track of the addresses. The cache

*This work was primarily supported by Cadence Design Systems (India) Pvt. Ltd. and done at Cadence Research Center at IIT Kanpur.

simulator is highly configurable wherein the designer can specify the caching policies. The cache specifications include the hierarchy level of the cache, cache type (instruction/data/unified), associativity, cache size, cache line size, replacement policy, write back policies etc. The cache simulator can simulate hierarchy of caches. There were a couple of reasons as to why we chose to use an on-line cache simulation as opposed to an off-line cache simulation (from the address trace). To perform cycle-accurate timing simulations, it is necessary to take into account cache delays. As a later project, we intend to implement a timing simulator. The present cache simulator could be integrated into this timing simulator. During instruction/data access, the cache simulator would estimate the delay in accessing the particular memory reference and provide the simulator the number of clock cycles the processor has to stall. Another reason to choose an on-line simulation policy was that trace generation consumes huge disk space and disk I/O time. Moreover, off-line cache simulation might be an inaccurate simulation of the address traces, as it does not take into account the variations in addresses due to instruction reordering by the processor. An on-line cache simulation is indeed able to handle this and hence there is a need to tie it to a retargetable functional simulator.

The rest of the paper is organized as follows, In section 2, we list the related work. In section 3, we give an overview of the *Sim-nML*. We describe the integrated simulation environment in section 4 and the cache simulator is described in section 5. We also explain the mechanism to integrate the cache simulator with the functional simulator and to simulate various caching mechanisms at run-time. Finally, we present some simulation performance results in section 6 and conclude this paper.

2. Related Work

Performance modeling of a system is a growing area and a lot of research has been pursued in this area. These previous works have resulted in a set of performance modeling tools focusing on various aspects.

Cheetah [12] is a cache simulation package which can simulate various cache configurations in a single pass through the address trace. Specifically, *Cheetah* can simulate ranges of set-associative, fully-associative or direct-mapped caches.

ATOM [11] provides a framework for providing customized program analysis tools. It provides a common infrastructure provided in all code-instrumenting tools. *ATOM* organizes the final executable where the application program and user's analysis routines run in the same address space. *ATOM* uses no simulation or interpretation. It has been used to build a diverse set of tools for basic block counting, profiling, dynamic memory recording, instruction and data cache simulation, pipeline simulation, evaluating branch prediction and instruction scheduling.

Pixie [10] is a utility that allows one to trace, profile or generate dynamic statistics for any program that runs on a MIPS processor. It works by annotating executable object code with additional instructions that collect the dynamic information during run time.

Dinero IV [2] is a trace driven uniprocessor cache simulator for memory reference.

QPT [4] [5] is profiler and tracing system. It rewrites a program's executable file (a.out) by inserting code to record the execution frequency or sequence of every basic block or control-flow edge. From this information, another program QPT_STATS can calculate the execution cost of procedures in the program.

EEL [6] (Executable Editing Library) is a C++ library that hides much of the complexity and system-specific detail of editing executables. EEL provides abstractions that allow a tool to analyze and modify executable programs without being concerned with particular instruction sets, executable file formats, or consequences of deleting existing code and adding foreign code. EEL greatly simplifies the construction of program measurement, protection, translation, and debugging tools.

SimOS [7] is a machine simulation environment designed to study large complex computer systems. *SimOS* simulates the computer hardware in sufficient detail and speed to run existing system software and application programs.

Visualization based Microarchitecture Workbench (**VMW**) [13] is an infrastructure which facilitates the specification of instruction set architecture and microarchitecture of a machine in a concise manner. *VMW* provides all necessary infrastructure software to the designer, including generic simulation software, visualization support software and graphical user interface software. VMW automatically integrates the machine specification and infrastructure software to generate a customized performance simulator based on the trace-driven simulation approach. Thus *VMW* provides a powerful environment for modern superscalar processor design.

3. Sim-nML

Sim-nML [9] is a direct extension of *nML* [3] machine description formalism. It includes several features which are useful for the performance simulation and are not present in *nML*.

Sim-nML is targeted for describing any arbitrary processor architecture at the instruction set level hiding implementation details. The instruction set is described in a hierarchical manner. The semantic actions of the instructions are captured as fragments of code spread all over the instruc-

tion tree. The *Sim-nML* specifications are described using an attribute grammar. There is a fixed start symbol called *instruction*, and two types of productions, *and-rule* and *or-rule*.

There are certain fixed attributes defined for *and-rules* which capture various aspects of the instruction set. The *syntax* attribute captures the textual assembly language syntax of the instructions. The *image* attribute captures the binary image of the instructions. The *action* attribute captures the semantics of the instructions. The *uses* attribute captures the resource usage model and is used for timing simulation.

The following illustration is a specification for a simple processor with four instructions – *add*, *sub*, *bim* (branch immediate), and *bin* (branch indirect).

The processor has two addressing modes – immediate and register indirect. The four instructions are hierarchically described. The branch immediate (*bim*) instruction modifies the PC with an immediate branch offset. The branch indirect (*bin*) takes the branch address in the specified register and puts it in the PC. The *add* instruction adds two registers and puts the result in the first register. The *sub* instruction subtracts two registers and puts the result in the first register. Retargetable tools flatten out the hierarchical description to enumerate out the complete instruction set and its associated attribute definitions. Different tools, depending on their need, use different attributes. More description of *Sim-nML* can be found in [9].

4. Integrated Simulation Environment

The goal of this work is to model a complete system environment with a processor core specified using *Sim-nML*. The processor simulator for a given application can be generated automatically from the *Sim-nML* processor description. Such a simulator, could be either functional or cycle accurate.

The functional simulator is essentially an instruction set simulator. The retargetable functional simulator generator - *Fsimg* [1] uses a processor description in *Sim-nML* and a program binary for that processor. It then generates an instruction set simulator capable of simulating the instructions of the input program. The instructions are simulated in sequential program order. The overall process is shown in figure 1.

Fsimg initially flattens the hierarchy in *image* and *action* attributes of the *Sim-nML* description. For each machine instruction, *Fsimg* emits a corresponding unique *C* function. This function is obtained by translating flattened *action* attribute definition to C. Fsimg then reads the program binary and for every matched input instruction image, it generates

```
const REGS = 5
type word = int ( 16 )
reg R [ 2**REGS, word ]
reg PC [ 1 , word ]
mode IMM ( n : card ( 12 ) ) = n
  syntax = format( "%d", n )
  image  = format( "%12b", n )

mode REG_IND ( n : card ( 5 ) ) = R [ n ]
  syntax = format( "r%d", n )
  image  = format( "%5b", n )

resource bu, alu
op instruction ( x : instr_action )
  uses = x.uses
  syntax = x.syntax
  image  = x.image
  action = {
             PC = PC + 2;
             x.action;
           }

op instr_action = branch_inst | arithmetic_inst

op branch_inst = bim | bin

op bim ( d : IMM )
  uses = bu #1
  syntax = format( "bim %s", d.syntax )
  image  = format( "1000%s", d.image )
  action = { PC = PC + (d << 4); }

op bin ( r : REG_IND )
  uses = bu #1
  syntax = format( "bin %s", r.syntax )
  image  = format( "10010000000%s", r.image )
  action = { PC = r; }

op arithmetic_inst = add | sub

op add ( r1 : card ( 5 ), r2 : REG_IND )
  uses = alu #1
  syntax = format( "add r%d %s", r1, r2.syntax)
  image  = format( "10100%5b%s", r1, r2.image)
  action = { R [ r1 ] = R [ r1 ] + r2; }

op sub ( r1 : card ( 5 ), r2 : REG_IND )
   uses = alu #1
   syntax = format( "sub r%d %s", r1, r2.syntax)
   image  = format( "10110%5b%s", r1, r2.image)
   action = { R [ r1 ] = R [ r1 ] — r2; }
```

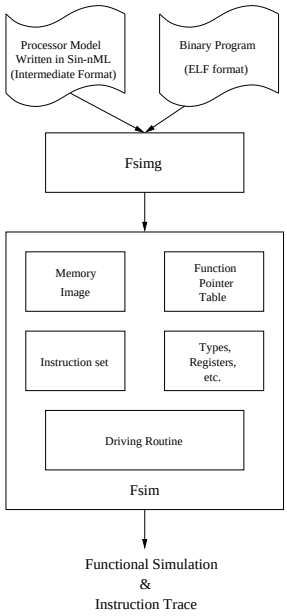

Figure 1. A View of Functional Simulation Process

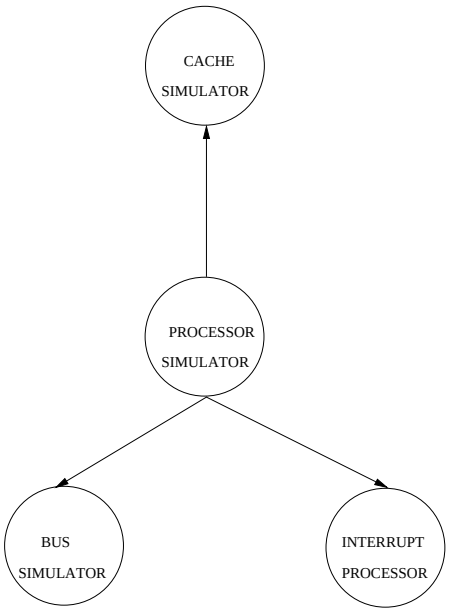

Figure 2. A View of Integrated Simulation Environment

a call to the corresponding function defining that instruction. Thus, it generates a list of function calls corresponding to all instructions in the input program. The driver routine of the functional simulator, simulates the program by calling functions sequentially until the program terminates.

Work is currently in progress to generate a cycle accurate microarchitecture simulator. This would use the *uses* attribute of *Sim-nML* to model the resource usage pattern of each instruction. A preliminary version of the cycle accurate simulator has already been developed [8].

The processor core simulator would form the heart of the integrated simulation environment. For the purpose of system simulation and evaluation, other components like the cache, bus, memory, interrupts etc. have to be simulated. Interface functions or *canonical functions* have been provided in *Sim-nML* to facilitate the interaction of the processor core with the external world. The designer is free to choose his own model to simulate the external environment as long as it adheres to the calling convention of the *canonical functions*. This provides a flexible and convenient way for system simulation. Currently, the cache simulation environment is designed as a *C*-function that is invoked by the processor simulator. The cache simulator then executes synchronously with the processor simulator.

The overall process is shown in figure 2.

5. Cache Simulator

Sim-nML was designed for specifying the instruction set and an abstract microarchitecture model of a processor. For a system design, it is necessary to model not just the processor but also other components of the system like the cache, memory, bus etc. This would facilitate evaluation of the combined performance of the system for a given application domain. *Sim-nML* provides a flexible mechanism for interface between the processor and other system components through the use of *canonical functions*. *Sim-nML* language specification does not impose any discipline on how the *canonical functions* are to be implemented. It is left to the tools to decide on the interface and implementation of the *canonical functions*.

The cache simulator integrates with the functional simulator through the *canonical functions*. For cache simulation, the *canonical functions* are specified in the *action* attribute of the *Sim-nML* specification (see example in following subsection). The functional simulator implements the *canonical functions* in a parametrized way as *C* functions. During its execution, the functional simulator calls these *C* routines. For the cache simulator, two canonical functions *icache* and *dcache* are used in the *Sim-nML* specification. They provide an interface to the cache simulator back-end. This back-end implements the instruction cache, data cache and the unified cache. The *icache* function call provides the cache functionality for the instruction access while *dcache*

function call provides the same for the data access. A separate cache configuration file is used to select the caching policy.

5.1. Instruction Cache Simulation

The *icache* canonical function call acts as the interface between the processor simulator and the instruction access functionality of the cache simulator. The *icache* canonical function call is inserted in the *action* attribute of the top level *instruction* node in the *Sim-nML* specification. This is done so that, during flattening of the *Sim-nML* specification, all instructions would contain calls to this interface function. The functional simulator implements *icache* as a *C* routine passing it the address of the current instruction as a parameter. For the example processor specification given in page 3, the *"icache"* call is inserted as shown below

```
op instruction ( x : instr_action )
    uses = x.uses
    syntax = x.syntax
    image = x.image
    action = {
        "icache"(PC);
        PC = PC + 2;
        x.action;
    }
```

Here, PC is declared as a register which holds the address of the current instruction. During simulation, the functional simulator calls the *icache* routine for each instruction. The instruction cache simulator keeps track of the instruction addresses and evaluates the cache hits and misses depending on the caching policy specified in the configuration file.

5.2. Data Cache Simulation

For data cache simulation, the *"dcache" canonical function* call acts as the interface between the processor simulator and the data cache simulation routine of the cache simulator back-end. The *'dcache' canonical function* calls are inserted in the *Sim-nML action* attribute for memory access instructions, typically load/store instructions. Just as the *"icache"* canonical function, the *"dcache"* canonical call is implemented in the functional simulator as a *C* routine. The *dcache* routine takes two parameters, the data memory address and the mode of memory access operation. This could be either a read or write to the cache. The *"dcache" canonical function* call is specified as shown below.

```
op load_byte ( x : 32bitaddr )
    uses = load_store_unit #3
    syntax = format("lb %32b", x)
    image  = format("1100%32b", x)
    action = {
        "dcache"(x, READ);
        A = M[x];
    }
```

The instruction *load_byte*, loads a byte from the memory *M* into the accumulator *A*. The *canonical function* call *"dcache"* is passed the address of the memory location from which a byte is to be loaded and also the mode of memory access, READ. The other access mode is WRITE which is used to specify store operations. This access mode is used in implementing the write-through and write-back caching policy. In the write-through policy, the data is written directly into memory simultaneously while writing into the cache. In write-back, only the cache is updated.

5.3 Unified Cache Simulation

Depending upon the cache configuration the *icache* and *dcache* functions can simulate the unified cache as well. For this, both functions essentially perform the same functionality.

5.4 Cache Specification

A cache configuration file is used to characterize the cache to be simulated. The cache simulation functions *icache* and *dcache* read the configuration file during initialization and creates the specified cache simulation environment. A sample cache configuration file is given below.

```
# Cache Configuration File

# Number of Levels
levels 1

# Address length
addrlen 32

# Description for L1 Cache
Level 1

# Description for InstrCache of L1
type INSTRCACHE
associativity 4
size 32K
line 16
replace FIFO
subblock 4 # subblock size
write WB WA # Write Back, Write Allocate
writebuffer 32 # size of write buffer
nonblocking 2 # number of outstanding misses

# Description for DataCache of L1
type DATACACHE
associativity 4
size 32K
line 16
subblock 4
replace FIFO
write WB WA
writebuffer 32
nonblocking 2
```

This following are the parameters used in the cache configuration file.

- **levels**: This specifies the total number of levels of cache. The first level is named as **L1**, second as **L2** and so on.
- **addrlen**: This parameter specifies the length of the memory address.
- **level**: This is used to define the level that is being described. **Level n** stands for the n^{th} level of cache in the hierarchy.
- **type**: This parameter defines the cache type. It could be **INSTRCACHE** for instruction cache or **DATACACHE** for data cache or **UNIFIED** for a unified cache architecture.
- **associativity**: This specifies the associativity of the cache being described. For a direct mapped cache the associativity is **1**. For an *n*-way set associative cache the value provided is *n*. A separate constant **FULL** is used for describing a fully associative cache.
- **size n**: It specifies the cache size, **n** can be suffixed with *K*(kilobytes) or *M*(megabytes). Without the qualifier, **n** is assumed to be in bytes.
- **line n**: This specifies the cache line size, **n** can be suffixed with *K*(kilobytes) or *M*(megabytes). Without the quantifier, **n** is assumed to be in bytes.
- **replace**: This denotes the replacement policy for *set associative* cache systems. The policy can be **FIFO** (first in first out), **RANDOM** or **LRU** (least recently used).
- **subblock**: This denotes the subblock size within a cache line.
- **write**: This specifies the write policy. It could be one of the following:
  - **WB WA**: write through with write allocate
  - **WB NWA**: write back - no write allocate
  - **WT WA**: write through with write allocate
  - **WT NWA**: write through with no write allocate
- **writebuffer**: This specifies the size of the write buffer in bytes.
- **nonblocking**: This specifies the number of outstanding misses that a cache can satisfy without blocking the processor.

The cache simulator assesses and accumulates the following parameters - cache hits, cache misses, conflict misses, invalid misses, compulsory misses. During simulation it internally keeps track of the above metrics. At the end of simulation, the statistics are dumped into log file for later analysis. Statistics are maintained for each cache type at each cache level.

We have been able to describe several of processors in *Sim-nML*. They include PowerPC603, Motorola68HC11, ADSP2101, Intel8085, ARM, Sparc. The cache simulation was extensively tested for PowerPC603 and Motorola processors. The description of these processors can be obtained at http://www.cse.iitk.ac.in/sim-nml.

Program	Description
mmul.c	Matrix multiplication program. This program initializes two integer matrices of 100x100 size and multiplies them.
bsort.c	Bubble sort program. This program initializes an array of 1500 integers in descending order and sorts them to ascending order using bubble sort algorithm.
qs.c	Quick sort program. This program initializes array of 1,00,000 integers in descending order and sorts them to ascending order using quick sort algorithm.
fmmul.c	Matrix multiplication for floating-point numbers. Initializes and multiplies two floating point matrices of size 100x100.
nqueen.c	This program finds all the possible ways that N queens can be placed on an NxN chess board so that the queens cannot capture one another. Here N is taken as 12.

Table 1. Benchmark Programs

Program	Total No. of Instructions
mmul.c	111,681,966
bsort.c	60,759,034
qs.c	80,773,862
fmmul.c	112,281,966
nqueen.c	204,916,928

Table 2. Total number of instructions simulated for test programs.

Cache	Description
cache-conf-1	8K unified cache, 32 byte line size, 4-way associative, LRU replacement policy.
cache-conf-2	8K instruction, data cache, 32 byte line size (both), 4-way associative (both), LRU replacement policy (both).
cache-conf-3	2 Levels, Level 1: 8K instruction data cache, direct mapped, Level 2: 32K instruction, data cache, 4-way associative, LRU replacement policy, Both levels have 32 byte line size.
cache-conf-4	3 Levels, Level 1: 8K instruction, data cache, 4-way associative, LRU replacement policy, Level 2: 32K instruction, data cache, 8-way associative, LRU replacement policy, Levels 1, 2 have 32 byte line size, Level 3: 64K unified cache, direct mapped, 64 byte line size.

Table 3. Benchmark Programs

6. Results

In this section, we consider the slow down of the retargetable functional simulator *Fsim* because of the cache simulator. The cache simulator is run in various configurations and the slow down in simulation speed is noted. Five benchmark programs were written in *C* (Table 1) and compiled for *PowerPC 603 Sim-nML* processor description. The simulators were run on a *Pentium-II 350MHz* machine.

Table 2 gives the total number of dynamically executed instructions during the simulation of each of the programs as reported by the functional simulator.

The performance results of functional simulator are given in Table 4.

Table 3 lists the different cache configurations which were used to run the cache simulator. Table 5 compares the performance of the functional simulator with cache simulation for each of these cache configurations.

From Table 6 we observe on an average of 80% slowdown in simulation speed because of online cache simulation.

Program	Target instructions per second. Fsim compiled with optimization level 3 (in million instructions per second)
mmul.c	.89
bsort.c	1.1
qs.c	1.3
fmmul.c	1.3
nqueen.c	1.4

Table 4. Simulation speed of the functional simulator without cache simulation

cache-conf	1	2	3	4
mmul.c	.22	.22	.29	.20
bsort.c	.20	.20	.20	.20
qs.c	.17	.17	.07	.16
fmmul.c	.21	.21	.28	.20
nqueen.c	.29	.29	.34	.28

Table 5. Simulation speed of functional simulator with cache simulation (in million instructions per second)

7. Conclusion

In this paper we presented an execution driven retargetable cache simulator. The cache simulator is modular with the only interface to the instruction set simulator being the *canonical functions*. Thus it is possible to change the processor description and hence the address trace without modifying the cache simulator. The cache configuration file provides a convenient and easily adaptable way to simulate multiple caching policies. The cache-simulator does not slow down the instruction set functional simulator to

cache-conf	1	2	3	4
mmul.c	75	75	67	77
bsort.c	81	81	81	81
qs.c	83	83	94	83
fmmul.c	83	83	78	84
nqueen.c	79	79	75	80

Table 6. Percentage decrease in simulation performance due to cache simulation

unacceptable range.

References

[1] S. Chandra and R. Moona. Retargetable functional simulator using high level processor models. *Proceedings of International Conference on VLSI Design, Calcutta, India.*, January 2000.

[2] J. Edler and M. D. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator.

[3] M. Freerick. *The nML Machine Description Formalism*. http://www.cs.tu-berlin.de/~mfx/dvi_docs/nml_2.dvi.gz, 1993.

[4] J. R. Larus. Efficient Program Tracing. *IEEE Computer*, 26(5):52–61, May 1993.

[5] J. R. Larus and T. Ball. Rewriting Executable Files to Measure Program Behavior. *Software Practice & Experience*, 24(2):197–218, Feb 1994.

[6] J. R. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 1995.

[7] S. D. Mendel Rosenblum, Edouard Bugnion and S. A. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, Jan 1997. http://simos.stanford.edu.

[8] V. Rajesh. A generic approach to performance modeling and its application to simulator generator. Master's thesis, Dept. of Computer Science and Engg., IIT Kanpur, India, http://www.cse.iitk.ac.in/research/mtech1996/9611132.html, July 1998.

[9] V. Rajesh and R. Moona. Processor modeling for hardware software co-design. *Proceedings of International Conference on VLSI Design, Goa, India.*, January 1999.

[10] M. D. Smith. Tracing with Pixie. *Memo from Center for Integrated Systems, Stanford Univ.*, April 1991.

[11] A. Srivastava and D. Wall. ATOM: A system for building customized analysis tools. *Proceedings of the SIGPLAN '94 Conference of Programming Language Design and Implementation (PLDI)*, pages 196–205, June 1994.

[12] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with application to miss characterizations. *Proceedings of the 1993 ACM Sigmetrics Conference on Measurements and Modeling of Computer Systems*, May 1993.

[13] D. Trung A. and S. John Paul. VMW: A Visualization-Based Microarchitecture Workbench. *IEEE Computer*, pages 57–64, Dec 1995.