Augmint - A Multiprocessor Simulation Environment for Intel x86 architectures [1]

Arun Sharma, Anthony-Trung Nguyen and Josep Torrellas
Center for Supercomputing Research and Development,
University of Illinois at Urbana-Champaign,
Urbana, IL 61081
{arsharma,anguyen,torrella}@csrd.uiuc.edu

Maged Michael
Department of Computer Science,
University of Rochester,
Rochester, NY 14627
michael@cs.rochester.edu

John Carbajal
Scalable systems division,
Intel Corporation,
Beaverton, OR
carbajal@ssd.intel.com

CSRD Technical Report 1463

March 28, 1996

[1]Thanks to Dave Archer, Intel Corporation

Abstract

Augmint is a fast execution driven multiprocessor simulator for Intel x86 architectures. It is based on MINT [1], but provides a user interface similar to that of Tangolite [2]. For the sake of simulation speed, processors are modelled as user level threads. A user defined memory hierarchy simulator can be plugged into the simulator to study the behaviour of the architecture under consideration.

1.1 Introduction

The design of the perfect memory hierarchy for scalable distributed shared memory architectures remains elusive, even today. Given that hardware prototyping and analytical modelling are either inflexible or inadequate, quite a few simulators have been written in the past to study the behaviour of parallel applications on these machines.

Simulating a multiprocessor architecture is a tradeoff between simulation time and the accuracy of the results obtained. The more the accuracy, the lesser the speed. There are simulation systems that perform cycle by cycle simulations, a method often used by hardware designers. But such simulations are extremely slow and are useful only in validating the architectures. Moreover, a simulator is also expected to be non-interfering, in order to get highly accurate results. By spending a lot of time in simulating the system, in a great level of detail, the timing accuracy of the simulator is compromised. Goldschmidt and Hennessy [5] deal with these issues at length.

Augmint deviates from the past work mainly in shifting the focus to CISC architectures. Most of the increased complexity was in writing the augmenter *Doctor* which now recognizes the complex and even implicit memory references in the Intel x86 assembly code.

Here is a brief overview of the rest of the report. Section 2 deals with the previous work and the various design choices. Section 3 provides an overview of Augmint. Section 4 describes the steps involved in building the final executable. Further, sections 5, 6 and 7 describe each of the pieces in Augmint in more detail. Lastly, section 8 sums up what has been done and where we are headed.

1.2 Design choices

There are several different approaches to multiprocessor simulation. One is based on interpreting the machine instructions and making calls to the memory system simulator on memory references. This approach in general is very slow, since interpreting one machine instruction takes 10 to 100 times more time than executing it natively. However, some simulators like MINT [1] have taken a hybrid approach by executing straight line pieces of code, without memory references natively and interpreting instructions with memory references.

A more common approach is to *augment* the code, that is, to insert additional instructions at the memory reference points to drive the simulator. This approach has been used in Proteus [6], Tangolite [2] and *Augmint*.

Such instrumentation can be done both at the executable level and the assembly level. Each method has its advantages and disadvantages. While augmenting the assembly code is more flexible, it has the disadvantage that the user has to get the assembly code for the library files, which need to be instrumented for the simulation to be accurate. On the other hand, instrumenting the executable, as has been done in the case of EEL [7] on the other hand does not suffer from this disadvantage. However, it needs a recomputation of the jump targets and labels.

Further, the simulation can be time driven, where the machine is simulated on a cycle by cycle basis or event driven, where the clock will be incremented based on certain events. By previous experience it has been found that the latter approach leads to more efficient simulation.

Finally, we examine trap driven simulation. In this approach, traps are set on memory locations by making use of error correction codes and the simulator takes control only during the traps. This approach has been used in WWT [8] and Tapeworm [9]. The advantage of trap driven simulation is that memory reference events, that do not miss in the simulated memory hierarchy do not cause a trap. As a result, the simulation is faster. However, since this approach is heavily dependent on the underlying architecture, it can not be used universally. Alternatively, traps can be set by manipulating the page tables. This approach is more portable, but suffers from the fact that page faults take several hundreds of cycles and hence the simulation performance suffers.

1.3 Overview

Augmint is essentially made up of the following components :

- **Doctor**, the augmenter, which takes as input an x86 assembly language file, inserts additional code at basic block beginnings and at memory reference points to drive the simulator and outputs an augmented assembly language file.

- **The threads package:** Large parts of the threads package from its predecessor MINT [1] remain unchanged. The most significant changes are due to, changing the object code interpretation to compile time assembly language instrumentation.

- **The user interface:** The simulator can be roughly divided into a *front end* and a *back end*. The *front end* simulates the execution of multiple processes running on multiple processors in parallel and generates certain *events* of interest. The *back end* is user-defined and invoked every time a significant event occurs. This interface is exactly the same as that provided by MINT [1].

1.4 Usage

There are several steps involved in building the executable, as shown in Figure 1. The application source code in which parallel constructs are expressed using the m4 macros is input to a macro processor, along with the macro library `c.m4.augmint`. A C compiler, in our case the GCC, compiles the resulting C code into x86 assembly code. In the next step, *Doctor*, the augmenter inserts appropriate code into the assembly file. The resulting file is then assembled and linked with the threads package and the architectural model to give the final executable. When the executable is run, it performs the simulation and prints the results on the standard output.

1.5 Doctor

1.5.1 Implementation details

Doctor uses the Unix compiler writing tools *LEX* and *YACC* to parse the input assembly file. Since an x86 instruction can make more than one memory reference, *Doctor* stores some state for each reference in a separate *reference record* inside the instruction record. [1] *Doctor* is guided by an input file, which gives the timing information and the possible number and the kind of memory references that each instruction could possibly make. This input file is automatically generated from the file `mnemonics.unix` by some *AWK* scripts.

On finding a memory reference, *Doctor* inserts code to pass its address and the size to the simulator. It also inserts code to update the clock at the

[1] Note that a single instruction can make more than one explicit/implicit memory reference

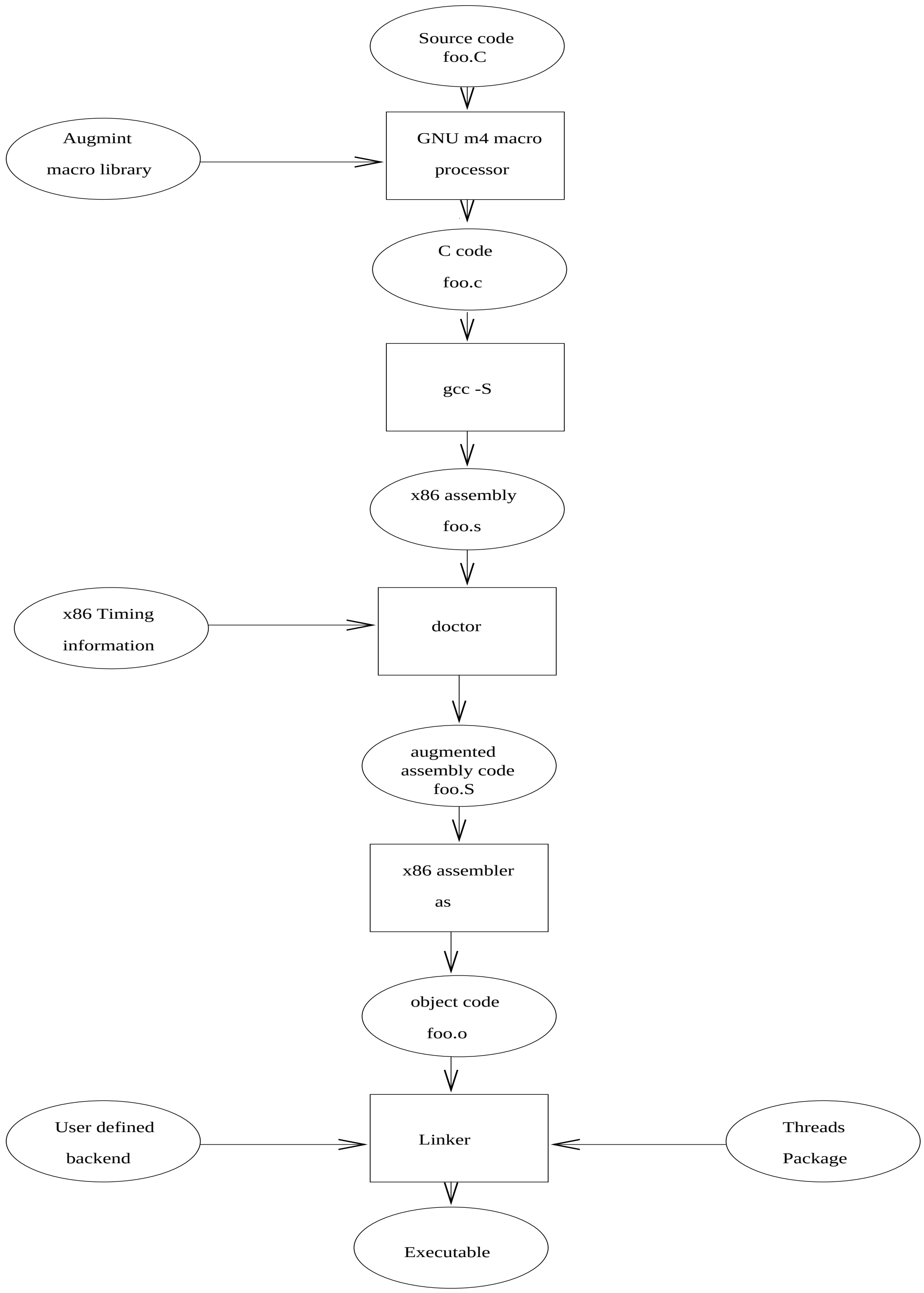

Figure 1.1: Compilation process

end of each basic block to account for the execution of instructions in that basic block.

Doctor also takes care of saving and restoring the CPU registers everytime the simulator is called, thereby ensuring that the whole operation is non-interfering.

1.5.2 Cycle counting

The number of cycles for each instruction is determined by the value in the input file `mnemonics.unix`. [2]. The cost of the memory reference is added to the thread clock, by the user defined simulator. This cost depends on whether the reference was a hit or a miss in the simulated cache hierarchy.

1.5.3 Data Movement

Augmint can be used to validate shared memory multiprocessor architectures. When used with the -V switch, it inserts code to actually move the data supplied by the backend into the memory location being accessed.[3] The correct execution of a good number of applications in the data movement mode is a good way of validating architectures.

1.5.4 Limitations

It is not possible to determine if some instructions in the x86 instruction set make a memory reference at compile time. These instructions, however, do not occur very frequently in the assembly code. Therefore the simulation is quite accurate.

1.5.5 Improvements

There are several possible optimizations to improve the performance of the simulator further. Currently, a lot of time is spent on updating the simulated clock, which is represented as a `double`. Such an update is done at the end of each basic block and before entering the simulator thread. A possible optimization, discussed in the Proteus [6] user's guide, is to merge basic blocks together into bigger basic blocks wherever possible.

Further, depending on the length of simulation time, the user may choose to use a smaller data type for the clock, to improve the simulation speed.

[2] The default values have been taken from the Pentium processor manual [10]

[3] Stack references are not augmented.

1.6 The threads package

The threads package consists of a simulator thread, which initially schedules one application thread, viz `appl_main`. When the initial application thread runs, it creates more threads. When these threads make memory references or do anything that is of interest, *events* are generated and the simulator thread gets the control. The simulator thread examines the event and schedules a *task* to handle the event.

Tasks are the active entities in *Augmint*, in the sense that the simulation algorithm repeatedly looks for new tasks to execute. When there are no more tasks, the simulation ends. Threads are passive entities and one or more tasks are scheduled on their behalf. The user-defined backend may also create new tasks to handle the events. The MINT [1] user's guide deals with this topic in more detail.

1.7 The backend

We have provided a very simple example backend along with the simulator. Functions `sim_read()` and `sim_write()` are the hooks that a researcher needs to plug his simulator into.

1.8 Conclusion

Augmint currently runs on x86 platforms runing Solaris[4] and Linux. A Windows NT[5] port is currently being worked on. *Augmint* combines the good features of both MINT [1] and Tangolite [2] and provides a software simulation environment that is fast, flexible and accurate.

[4] Solaris is a trademark of Sun Microsystems Incorporated.

[5] Windows NT is a trademark of Microsoft Corporation.

Appendix A

Doctor

A.0.1 Commandline

```
doctor [ -cgV ] [ assembly-language-file ]
```

A.0.2 Description

Doctor is a 80x86 assembly code augmenter, designed to work with *Augmint*, a multiprocessor simulation environment. It takes an 80x86 assembly language program, generated by a C compiler, such as GCC and analyzes it for memory references. Whenever it finds a memory reference, depending on the command line options, it inserts additional code, just before or after the instruction which makes the memory reference. This code saves the operand size, the virtual address and the type of the event and calls the simulator. Events can be of the following types: reads, writes and atomic read-modify-writes.

If no filename is given on the command line, *Doctor* reads the input from stdin and writes the output to stdout.

A.0.3 Options

Doctor accepts the following options.

-c Count references. With this option, *Doctor* merely counts the number of memory references - both reads and writes to the memory.

-g This option causes *Doctor* to instrument global memory references. Note that stack references are not augmented.

-h Prints a usage message.

-V Verify the memory simulator. With this option, data is actually moved to and from *Augmint*

Bibliography

[1] Jack E. Veenstra and Robert J. Fowler. MINT Tutorial and User Manual, Technical Report 452, The University of Rochester, Computer Science Department, August 1994.

[2] Stephen R. Goldschmidt. Simulation of Multiprocessors: Accuracy and Performance, PhD Thesis, Stanford University, June 1993.

[3] Stephen Alan Herrod. Tangolite: A Multiprocessor Simulation Environment, Stanford University, 1993.

[4] Stephen R. Goldschmidt and Helen Davis. Tango introduction and tutorial. Technical report CSL 40-910, Stanford University, January 1990.

[5] Stephen R. Goldschmidt and John L. Hennessy. The Accuracy Of Trace-driven Simulations Of Multiprocessors Stanford University technical report: CSL-TR-92-546, September 1992.

[6] Brewer, Eric A. Proteus: a high-performance parallel-architecture, MIT. Lab. for Computer Science MIT/LCS/TR 516, September 1991.

[7] James R. Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing (SIGPLAN '95 Conference on Programming Languages Design and Impelementation (PLDI), June 1995).

[8] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers, (ACM SIGMETRICS, May 1993).

[9] R. Uhlig, D. Nagle, T. Mudge, and S. Sechrest. "Trap-driven simulation with Tapeworm II," 6th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), Oct. 1994, pp. 132-144.

[10] Pentium processor manual