Armadillo: A High-Performance Processor Simulator

Brian Grayson and Craig Chase

TR ECE-PDS-1996-008 April 1996

Parallel & Distributed Systems group
Department of Electrical & Computer Engineering
University of Texas at Austin
Austin, Texas 78712

Armadillo: A High-Performance Processor Simulator*

Brian Grayson† Craig Chase
bgrayson@ece.utexas.edu chase@ece.utexas.edu

Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084

April 18, 1996

Abstract

Armadillo is a high-performance, execution-driven, non-native multiprocessor simulator. Each node of the simulated architecture is highly configurable, including the configuration of the superscalar, PowerPC-based CPU and many aspects of the memory hierarchy (cache, busses, memory, and the network interface). This paper discusses the design and implementation of the simulator, and also presents some single-node simulation performance results.

1 Introduction

1.1 Objective

The goal of this research was to create an object-oriented, high-performance, easily-reconfigurable processor simulator. The simulator should also be easily extendable to simulate a multiprocessor, in order to provide a suitable virtual architecture for future research topics. The performance goal was to achieve 10,000 simulated instructions executed per second.

Armadillo is a high-performance multiprocessor simulator being developed to study and evaluate parallel architecture design tradeoffs, and meets all of the above goals. The simulator has been carefully designed to provide easy portability to both new host processors and to a multiple-processor platform (either LAN- or bus-based). Armadillo's modular design allows the simulated architecture to vary greatly, from a single node with merely a CPU and memory, to a configuration as complicated as a multiprocessor of multiple nodes, where each node has several CPUs sharing memory or a bus. The simulated multiprocessor will be capable of operating under a message-passing or a shared memory communication paradigm.

Armadillo is execution-driven and fully-interpretive, so the configuration of the virtual CPU can be modified to increase the virtual architecture's performance. The virtual CPU is superscalar, and supports multiple integer pipelines and out-of-order execution and completion. While some processors perform in-order execution for instructions that require the same pipeline (the reservation stations are FIFO queues), the virtual CPU in Armadillo does not, but still preserves the appearance of sequential execution via full on-the-fly data dependence analysis. In other words, the dispatch queue is really an unordered "pool," and instructions are dispatched from the pool only when all data dependences have been satisfied.

Armadillo also provides determinism, so repeated experiments behave identically. This

*An earlier version of this work appeared as the Master of Science Thesis for the first author.

†Brian Grayson is supported by a National Science Foundation Graduate Fellowship.

greatly simplifies debugging of the simulated program and of the simulator.

Since the multiprocessor aspects of the simulator are meant to be modified for future research, this paper will concentrate on the single-processor simulation aspects of Armadillo.

1.2 Organization of this Thesis

Chapter 2 discusses various processor simulation techniques. Then, Chapter 3 describes the virtual architecture that is simulated by Armadillo, and Chapter 4 provides details about using the simulator. Chapter 5 relates aspects of how the simulator works. Next, Chapter 6 discusses trace-generation and debugging features in the Armadillo simulator. Chapter 7 presents some performance results, and Chapter 8 summarizes the paper.

1.3 Terms

Throughout this paper, the term "virtual architecture" refers to the simulated architecture, which for Armadillo is PowerPC-based. "Virtual processor" and "CPU" refer to the simulated PowerPC processor. The terms "host architecture" and "host processor" refer to the actual processor on which the simulator is executed.

2 Background

2.1 Why is Processor Simulation Useful or Necessary?

Processor and architecture evaluation can be done either through the use of hardware prototypes or through software simulation. Software simulation of processors is both useful and necessary, for many reasons. Simply put, software simulation is necessary to keep costs low, and simulation is useful for the extra flexibility and inspectability it can provide.

Software simulation keeps the financial, time, and designer-effort costs down. First, hardware prototyping requires a large financial investment for constructing and debugging the prototypes. Also, since building prototypes takes time, the latency from one version of the prototype to the next is quite large. In addition, it takes much more effort from designers to construct a modified hardware schematic ready for fabrication than it does for a simulator to be modified – everything, down to new masks for fabrication, needs to be done, versus modifications and a recompile for software simulation.

In addition, software simulation can provide simulation features that are either not readily available, or not available at all, at the hardware level. For example, software simulations allow the designers to have a greater degree of inspectability. If details about a submodule are desired, counters and other code can be added to the submodule to collect the desired statistics. With hardware, similar probing and data-gathering would require sophisticated hardware and software to achieve the same effect. Also, software simulation provides greater flexibility than hardware prototyping. Parameters can be modified, with relatively instantaneous feedback. Clock frequencies can be simulated at higher speeds, whereas hardware prototypes would have to be made with cutting-edge technology rather than cheaper technologies. Lastly, complicated hardware, that currently is too costly to use but should be affordable once the chip goes to market, can be simulated at a lower cost.

2.2 Simulation of processors

There are several different techniques that can be effectively used in a university research environment to simulate processors. These techniques include hardware emulation, fully-interpretive software simulation, direct-execution (or augmentation), and trace-driven simulation.

Hardware emulation of processors involves creating a special-purpose machine, and can provide details at the RTL level or behavioral level. RPM [1] is an example of a hardware emulation system. Each physical processor typically emulates a single virtual processor.

Fully-interpretive (also called program-driven or instruction-driven) software simulation completely simulates all relevant details of the simulator, usually at the behavioral level. Armadillo and CacheMire [3] are examples of processor simulators that are program-driven. In [5], Cragon presents a processor specification (in LISP) that is directly executable, which could be considered a fully-interpretive verifier (rather than simulator).

Direct-execution simulation (also called execution-driven simulation) requires a physical processor with an architecture similar to the simulated architecture. The simulated program is run directly on the processor (no simulation), except for a small subset of instructions (memory or network code) that trap to the simulator for detailed simulation. Examples of recent direct-execution simulators include Tango/Tango Lite [6, 9], the Wisconsin Wind Tunnel [10], and PROTEUS [2].

Trace-driven simulation restricts the simulated instructions to only the instructions of interest, and doesn't simulate the remaining instructions. This leads to better simulator performance, but limits the amount of detail that can be provided about the virtual architecture. For example, a trace-driven cache simulator usually ignores arithmetic instructions, so the interactions between calculation and memory accesses is restricted to the original interactions stored in the trace.

Each of the methods for processor simulation has advantages and disadvantages, many of which are summarized below.

2.2.1 Hardware Emulation

The primary advantage of hardware emulation is performance. A well-built hardware emulator will usually perform simulations in much less time than a fully-interpreting simulator, and may be faster than minimal-detail trace-driven and direct-execution simulators. For example, the first RPM emulator [1] can simulate a clock cycle in 8 physical clock cycles. This slowdown factor, and the emulator's clock cycle time, is expected to be reduced in later versions.

Another benefit of hardware emulation is that the emulator can function as a rough hardware prototype, since the emulator is usually constructed through VHDL or RTL models. This may help expose design problems or reinforce architecture decisions.

Hardware emulators, though, are typically costly to design and operate. If the emulator chips are custom-designed and fabricated, the costs are extremely high. Even when programmable logic devices (PLDs) are used, the research group still needs to invest in device programmers, peripheral supplies (interconnect medium, hard drives, support chips), and debugging devices (digital oscilloscopes, digital simulators).

Besides the hardware costs, the additional difficulties of modelling a processor via VHDL or RTL versus a higher level language like C or C++ may generate a large amount of complexity and frustration. This frequently translates into longer delays and more debugging work.

Hardware emulators also don't financially "scale" as easily as software simulators. To simulate a parallel processor with twice as many nodes, one would need to double the number of hardware emulator nodes. For software simulators, one can simply run multiple simulation processes on a single CPU. (The performance doesn't scale as well in software simulation, but most university research environments are willing to trade performance for cost.)

2.2.2 Fully-interpreted Simulation

Fully-interpreted simulation has as its primary advantage its lack of restriction to existing hardware. This manifests itself in the simulator's high degree of accuracy, flexibility and inspectability.

Like hardware emulators, fully-interpreted simulators model the full interactions between all of the architecture's components. Effects, if any, due to changes to the architecture will be visible. Both direct-execution and trace-driven simulators sacrifice this high level of accuracy in order to achieve higher performance.

All aspects of the processor and its support devices can be modified to any degree. Simple modifications, such as changing the number of pipelines, the depth of the pipelines, or the latency of cache and network accesses, can all be changed with a reconfiguration and recompilation. With hardware emulators, some of these changes would require reprogramming the entire set of devices, and others are impossible. In addition, more drastic changes like simulating future generations of processors, wider register files or busses, and novel or expensive networks (fully-connected, for example) can be simulated without major cost.

Linked with the ability to accurately simulate superscalar processors is the ability to gather meaningful statistics about the internal workings of the processor. With a fully-interpreting simulator, "counting" code can be easily added to the modules under study. Direct-execution simulators, however, must separate the internal effects of the virtual computation and the simulator computation, both of which are performed on the physical processor.

Fully-interpreting simulators (provided they are written in a portable manner) can be executed on nearly any hardware platform. Thus, when a faster machine becomes available for simulation work, one can use the simulator on the new machine.

Also, a fully-interpretive simulator is (or can be designed so that it is) deterministic. Multiple runs of the same program with the same data should provide identical results, cycle for cycle. With a direct-execution simulator, one may encounter cache miss differences due to multiprocessing, swapping, or general I/O on the actual processor. This kind of variation may make certain debugging difficult due to the non-repeatability of a run.

Poor performance is the first disadvantage of fully-interpreted simulators. Since all instructions are decoded and simulated in full, even the most efficient simulator will suffer from a slowdown factor in the hundreds or even thousands. The three other simulation methods will usually outperform any fully-interpreted simulator.

Another disadvantage is that a large amount of the simulator must be complete before even toy programs can be run. If the simulator does not yet model loads or stores, for example, very few programs to check the correctness of the simulator can be run. Similarly, some sort of operating system shell (*e.g.,* a simple `libc` and some I/O) must be constructed. Trace-driven simulators do not suffer from these problems, since only a subset of the instructions are simulated. Also, direct-execution simulators and hardware emulators can allow the virtual processor to use the physical operating system.

2.2.3 Direct-execution Simulation (or Augmentation)

Direct-execution simulators typically have very low slowdown factors, and thus very high performance. Direct execution simulation uses the same virtual processor architecture as the physical processor, so long pieces of code in the simulated program can be run "natively" on the physical processor. Detecting which sequences of code can be run in such a manner is done during the compilation phase of the program to be simulated.

The Proteus authors [2] claim a typical slowdown of between 35 and 100, and a best-case slowdown of merely 2. Tango [6] had slowdowns ranging from 2 to 1,000 or more, depending on the application, while Tango-Lite [9] had a slowdown of around 750 for some applications. Armadillo, in contrast, has a slowdown factor between 3,000 and 10,000, depending on the instruction mix of the simulated program.

Direct-execution simulators also require less from the simulator itself. Only a small subset of instructions are simulated, so the simulator writers do not spend as much time modelling hardware in the simulator software.

This is offset, of course, by the programming issues involved in having the simulated program jump in and out of the simulator. The original program needs to be "augmented" at the assembly-language level with cycle-counting instructions for direct-execution basic blocks, and the simulated instructions need to be replaced with traps to the simulator. In addition, it is desirable that the extra code that is used to track performance and trap to the simulator should not have any effect on the simulation. In practice, this is difficult to guarantee – the code may cause extra cache flushes, require different register allocations, affect timing, or have other effects. Preventing or accounting for these effects requires further code overhead, and some effects, due to the architecture of the simulator or its host processor, may be impossible to avoid.

Also, in a multiprocessor simulation, long directly-executed blocks of code (especially loops) need to be broken up with no-op traps to the simulator. If these traps were not present, one processor could run ahead of other processors that were bogged down, perhaps by numerous cache misses. This discrepancy in running rates will force later synchronization operations to be very expensive, and could skew statistics gathered from the simulation. These additional no-op simulator checks will prevent any one processor from running too far ahead, but they will also have an effect on the simulator's performance, depending on the application.

The fact that the directly-executed code and the simulated code both execute on the same physical processor leads to differences between the actual simulated machine and the ideal simulated machine, especially for simulated superscalar processors. For example, if only memory operations are being simulated, the interactions between the load/store pipeline and the other pipelines can not be modelled in a direct-execution simulator. A cache miss load in a superscalar processor may not create stalls in the processor if sufficient integer and other instructions can be executed out-of-order, leading to an effective miss penalty of zero cycles. On the other hand, it could be that all later instructions depend on the result of the load, leading to a full miss penalty. With direct-execution, the cache miss (or the load/store instruction itself) will cause a trap to the simulator. The subsequent execution of the simulator will then not know how long of a penalty should be assessed for the cache miss, without examining the surrounding code and performing data-dependence analysis. In addition, by the time the simulator has finished, any dependences between instructions that were already in the execution pipelines before the cache miss and instructions that followed the load instruction will now be satisfied, since the other pipelines are definitely emptied by now. In an accurate simulation, it may be that those now-satisfied dependences would not have been satisfied by the time the next instructions were decoded, leading to further stalls. The importance of such details, of course, depends on the accuracy desired from the simulator, but all of these details can be accurately modelled by a fully-interpretive simulator or by a hardware emulator.

The program code to be executed on the virtual architecture also suffers a significant expansion from the augmentation, by a factor of 2 or 3 in Tango [6], and as high as a factor of 4 or 5, plus a constant addition of 9 MB (primarily for cache tags) [8] in Tango-Lite, depending on the memory simulator used. For large programs, this may limit the amount of memory available for data used by the simulated program.

In addition, the assembly-level augmentation required to achieve efficient, low-side-effect transitions from direct-executed code to simulated code will most likely be quite different from processor to processor. In order to port the simulator to a different processor, a different augmenting program will need to be created.

Lastly, the simulated architecture is limited to being similar to an available processor or multiprocessor architecture (depending on whether the simulator is run on a uniprocessor or multiprocessor system).

2.2.4 Trace-driven Simulation

Trace-driven simulation usually has an even higher performance than direct-execution simulation, since only the instructions of interest are simulated. No register-file values or condition code flags need to be tracked. Typically, trace-driven simulators only simulate the memory accesses or network operations, which usually account for a small percentage of the program, but generate the largest delays in a processor or multiprocessor system, and are thus the key instructions for performance measurements. By not simulating the non-interesting instructions, large speedups are possible.

The chief advantage of trace-driven simulation is also its chief disadvantage – it does not simulate in great detail, so many interactions between the processor and the memory and network can not be explored and modified. For example, one can not perform experiments such as changing the cache characteristics and seeing what effect that has on the CPU pipelines' utilization.

In addition, the trace files need to be generated in some way. This limits the kinds of architectures or processors that can be simulated. If the traces are generated from a working machine or a direct-execution simulator, it must have the same general architecture as the simulated architecture, which limits the trace-driven simulator to simulating virtual architectures that are minor variations on existing architectures. If on the other hand the traces are generated from a fully-interpreted simulator, then someone has to construct the fully-interpreted simulator.

2.2.5 Justifications for our Construction of a Fully-Interpreting Simulator

For Armadillo, we chose to write a fully-interpreting simulator. The basic reason is because this choice gives us the most flexibility in all aspects of the simulation. We can:

- gather accurate data at any level,
- modify completely every behavior of the processor and any of its support components,
- experiment with novel processor or support-component ideas,
- quickly port Armadillo to any architecture, and thus perform experimentation using the fastest machine accessible to us.

All of these features are expected to be used in our Armadillo-based research.

2.3 Why PowerPC?

Although many architecturally sound processors are available, we chose to use the PowerPC as our virtual processor architecture for several reasons. First of all, we wanted to be able to use available compiler tools, so a commodity processor should be chosen. A superscalar processor was desired, so that we could simulate modern and next-generation multiprocessor architectures, and investigate how much the latency-hiding features of superscalar processors affected network operations. Also, RISC processors are typically easier to simulate than CISC processors, so processors such as the Intel x86 processor family were not chosen. We wanted to avoid the complications of simulating register windows and delay slots, which eliminated the SPARC processors from consideration. Lastly, we wanted to have access to some workstations that used our chosen processor, so that we could use its native tools during the early phases of simulator construction to generate test programs and verify results.

The PowerPC family met all of these restrictions, and did not present any undesirable difficulties, so its instruction set was chosen as our basis.

2.4 Intended Future Research

The simulator was developed with the end goal of investigating low-overhead communication (LOCO) in communication protocols in parallel computers. Several papers [4, 13] have discussed how much of the overhead is due to software layers added to both sides of message passing (sending and reception) in various existing architectures. However, it may be possible to modify an architecture to provide better performance, primarily by moving some of the overhead to hardware rather than software. In order to investigate such topics, we need a powerful, modifiable, highly-inspectable parallel-computer simulator upon which real experiments can be run.

Armadillo, once it has been further developed from a uniprocessor simulator to a multiprocessor simulator, should provide a simulated architecture that matches all of these needs.

3 The Virtual Architecture

Armadillo currently can simulate a single node of a planned multiprocessor. Several diagrams of possible virtual architectures of a node are shown in Figure 1. A node has a CPU and a memory hierarchy. The memory hierarchy contains at least a memory, and may also contain Caches, a network interface (NIF), and several busses to connect them. Each of these subcomponents is called a module. The remainder of this chapter presents more detailed descriptions of the modules of the virtual architecture.

3.1 PowerPC CPU Units

Each node has at least a single CPU. (The simulator allows a virtual architecture with clusters of processors at each node). The CPU is based on the PowerPC User-level Instruction Set Architecture (UISA) [15].

The simulated CPU is intended to represent state-of-the-art or beyond, to allow meaningful work on computers that are possible using the latest technology. As such, the CPU is superscalar. The simplest configuration has 3 pipelines (integer, load/store, and the branch unit). A more advanced configuration allows multiple (no hard-coded limit) integer pipelines. Support for floating-point instructions is not planned at this point: we are mostly concerned with the network and memory subsystem, and floating-point workloads (for timing purposes) can be approximated using corresponding integer operations.

The processor is also a RISC (Reduced Instruction Set Computer) chip, which simply means all instructions are a uniform size (32 bits), each instruction performs a relatively simple operation (one that typically can be done in a few cycles), and all memory accesses are through load or store instructions. Besides being prized in many circles for performance, a RISC chip is easier to simulate, for the same reasons that give it good performance (uniform instruction size, simple pipelineable operations, modest but sufficient memory addressing mechanisms).

By allowing multiple pipelines, we are increasing the amount of instruction-level parallelism that can be exploited by the processor. Instruction execution follows a dataflow pattern, with only data dependences or resource dependences limiting the processor. This parallelism is detected automatically by the CPU from the standard (sequential) PowerPC instruction set.

3.1.1 Instruction Execution

The virtual CPU is able to decode multiple instructions per cycle. Dependence analysis is performed at this point, to preserve proper sequential semantics (all future phases of instruction handling allow reordering). The number of instructions decoded per cycle can be set at simulator compile time, and has no upper limit. Values above 5 are not very realistic with today's technology, but higher values prevent the decode stage from being a bottleneck.

Once an instruction has been decoded and its operand availability has been determined, it is moved into a dispatch "pool." If all of its operands are available, the instruction can be dispatched to its appropriate functional unit, even if earlier instructions have not yet been dispatched. Otherwise, the instruction waits in the "pool," with its missing operands forwarded to it by the producing instructions when they complete. Of course, multiple instructions can be dispatched from the pool to the appropriate functional units in one cycle (as with most constants in Armadillo, the limiting number of dispatched instructions per cycle can be set at compile-time, and has no upper limit).

Note that for the memory operations, a register-based data dependence may not exist between instructions that have a memory-based data dependence. Thus, the decode ordering of memory operations is also recorded, for later use in the load/store functional unit.

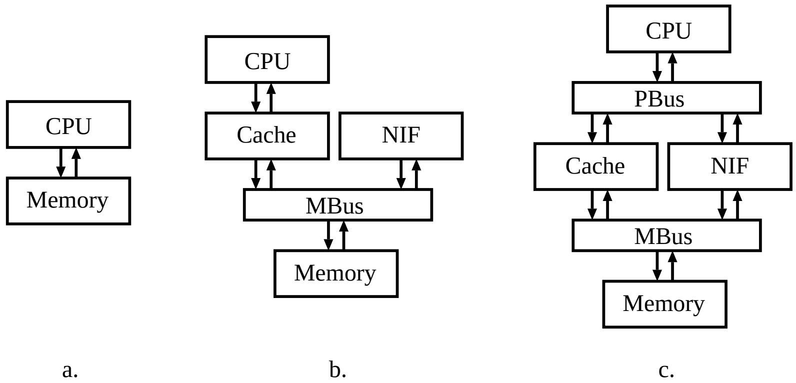

Figure 1: Sample Virtual Architectures

Each functional unit has at least one pipeline within it (the integer functional unit may have multiple pipelines). When an instruction is dispatched to a functional unit, all of its operands have already been loaded, so the instruction is placed in the FIFO queue for that pipeline.

Once an instruction is ready to complete, it forwards its output operands to the register file (unless the target register has been "renamed" since this instruction's decoding), and to any instructions in the dispatch pool that are waiting for its results.

Note that the Armadillo virtual CPU does not support precise interrupts (in fact, the CPU does not support interrupts at all). Thus, completions are allowed to take place out-of-order. If precise interrupts were desired, the CPU could be modified to contain a completion queue. However, this would slightly impair the virtual CPU's performance, and planned experiments will use polling techniques.

3.1.2 Memory-access Reordering and Split-phase Memory Transactions

One example of how the simulated node has been designed for maximum parallelism is the load/store pipeline. The load/store pipeline has been designed to be as aggressive as possible with regard to identifying and utilizing low-level parallelism. We now discuss the load/store pipeline in greater detail.

Each load or store operation has its dependences checked against all preceeding operations that have been decoded but have not yet completed (any previous instructions that *have* completed have by definition made their effects known to the entire system). If there are no dependences, the memory operation is dispatched immediately, perhaps out-of-order with respect to other reads or writes. Otherwise, it is distributed as soon as all of the dependent instructions complete. This reordering of memory operations (while still maintaining data dependence) avoids many "stalls" of the load/store pipeline that would otherwise occur.

The virtual CPU of Armadillo also supports split transactions. This splits, for example, a bus read into a read request and a read reply, with the bus free for other messages between the request and the reply. By allowing split transactions, the bus is available during long-latency activities (such as cache misses), and throughput is increased.

3.2 Memory Hierarchy

Each node also has a customizable memory hierarchy. The simplest configuration is one of just a memory module that can directly handle requests. A more complicated configuration has a bus between the processor and Cache (to allow for future cache-sharing multiprocessors), a Cache, another bus between the Cache and Memory (to allow for multiple caches, or multiple processors with private caches), and multiple Memory modules (*i.e.,* memory banks).

As the simulator continues to be developed and improved, it is expected that all modules in the virtual memory hierarchy will handle both receiving and generating split transactions. This increases the effective bandwidth at each bus interface, and thus of the whole system. Also, all modules will be able to handle multiple outstanding requests from every bus member, in order to utilize more parallelism available in the code.

3.3 Network Interface (NIF)

The network interface (NIF) is one of the key aspects of our planned research. The prototype NIF has memory-mapped control registers and buffers, and snoops MBus transactions. Various design options, such as including a number of intelligent buffers in the NIF, or providing it with DMA capabilities, will be explored in the future. In addition, the NIF may be placed at different levels in the memory hierarchy (in front of the Cache from the PBus, as a PBus peer with the Cache, or as an MBus peer with the Cache, for example).

The network interconnections themselves will be modelled exactly (hop-by-hop) or analytically, depending on the accuracy desired.

3.4 Library/Operating-System details

Some amount of operating system and `libc` support is required to simulate useful programs. Currently, very little `libc` and OS support is provided, because simulated programs are planned to use very few operating system features besides interprocessor communication.

The custom version of `libc` has a few functions like `getchar()`, `putchar()`, `malloc()`, and `exit()` (and derived functions such as `printf()`), and will be expanded as needed. System calls to the operating system are modelled by extending the PowerPC UISA and by using the `sc` (system call) instruction to trap to the simulator.

4 Using Armadillo

4.1 Host Requirements

The requirements for running the Armadillo simulator are quite simple: sufficient support for templates in C++ (such as that provided by `gcc 2.7.0`), to compile the simulator, Armadillo.

The requirements for generating executable programs for Armadillo to simulate are discussed in Section 4.3

Armadillo was developed on a Pentium host (little-endian, 32-bit words), and has been tested under both the NetBSD and Linux operating systems on that platform. In addition, Armadillo has also been tested on a Sun SPARCserver 20 (big-endian, 32-bit words), and on a DEC Alpha (little-endian, 64-bit words).

The PowerPC development tools (cross-compiler, cross-assembler, and cross-linker) were the GNU tools on the Pentium platform. Native RS6000 tools were used in the early stages, and occasionally throughout development for verification purposes.

4.2 Configuration of Armadillo

In order to create an Armadillo simulator, a configuration file that describes the virtual architecture must be created, listing the desired modules, various parameters for those modules, and the connections between the modules via ports. The configurator generates a C++ stub file for each module, and a Makefile for that configuration.

The use of a configuration file to describe the virtual architecture, with the use of modules and ports, allows the simulator builder to configure and create simulators without knowing about the internals of the various modules being used. In addition, as long as two implementation of a specific module (or even implementations of two different modules) both satisfy the expectations of other modules to which it is connected, either module implementation can be exchanged for the other by the simulator builder without modifying any code. This is what allows the simulator builder to change the memory hierarchy so easily – both the memory module and the cache module can be directly connected to the processor's memory ports and correctly handle the processor's requests.

Figure 2 shows a slightly simplified configuration file, for the base virtual architecture presented in Figure 1a. The description file contains two major parts. The first part, indicated by the keyword `modules`, describes the modules and their parameters. Here, there are two modules, a processor and a memory. The items in the first set of parentheses describe the directory in which the module code is found (for the processor, this is "`CPU`"), a header file to be used in generating a C++ stub for the module ("`cpu_make.h`"), and the name of the class representing the module ("`CPU`"). The strings in the second set of parentheses provide the names of the Ports for the module. The CPU has a `to_mem` and `from_mem` Port, while the memory module has `to_proc` and `from_proc` Ports. The remaining items inside the curly braces are used by the configurator to specify "include" directories and compile-time constants that are needed by the C++ module stub. The CPU module's code requires the `PPC` (PowerPC) and `INTERFACES` subdirectories to be on its C++ stub's include-file search path, and the memory module uses the `SIMPLE_READ_LATENCY` and `SIMPLE_WRITE_LATENCY` values to set its delays (in units of simulation cycles) for transactions.

The second section of the configuration file is denoted by the `connections` keyword. This section pairs an output port on one module with the corresponding input port on a different module.

After the configuration program has parsed the description file and generated a Makefile and C++ stubs in a "build" directory, one can then compile the Armadillo simulator in that directory by using the "make" command.

The configuration program can also handle arrays of modules and aggregation. This allows one to use the sample description above, wrapped by the array operator, to specify a multiple-node configuration by only adding a few lines to the description file.

Note that, contrary to direct-execution simulators, a single Armadillo simulator can run any program. With a direct-execution simulator, a custom simulator needs to be created for each executable to be simulated.

4.3 Generating PowerPC Executables for Simulation

The simulator accepts one common case (statically linked, no run-time relocation needed) of AIXCOFF executables (*e.g.,* generated from RS6000 workstations) for simulation on the virtual architecture. Thus, it can simulate AIX programs if they have been statically linked and rely completely on their own code (no system calls, no non-UISA code). These programs would of course have limited usefulness, so the libraries and operating system shell mentioned above have been developed.

The simulator was developed and tested using `gcc 2.7.0` and `binutils 2.6` (both obtained from the GNU/FSF project) to generate executables for simulation. These cross-development tools were run on a Pentium-based Unix system (NetBSD). Thus, access to an RS6000-based or PowerPC-based workstation is not necessary for any stage of the simulation process, even to generate executables for simulation.

5 The Armadillo Simulator Architecture

The Armadillo simulator is an event-driven simulator, with a global event queue. All events contain a timestamp that prescribes at what simulated cycle the event should occur. More than one event can be scheduled at a given cycle, in which case the relative order of the actions is FIFO. Also, some virtual times may have no events (for example, if the whole system is stalled on a cache miss).

```
modules
  proc(CPU, cpu_make.h, CPU) : (to_mem,from_mem)
  {
    include CPU/PPC
    include INTERFACES
  }
  mem(MEMORY, Mem_make.h, Memory) : (to_proc, from_proc)
  {
    SIMPLE_READ_LATENCY = 4
    SIMPLE_WRITE_LATENCY = 4
  }

connections
# connect processor and memory
proc:to_mem => mem:from_proc
mem:to_proc => proc:from_mem
```

Figure 2: Configuration File for a Simple Virtual Architecture.

Armadillo is written in an object-oriented fashion. Each block in the system architecture diagram, Figure 1, corresponds to a different module in the simulator. By creating these distinct boundaries, the modules can be used as building blocks, thus the interconnection between the modules can be easily changed via a configuration program, as mentioned in Chapter 4.

The Port mechanism is used for communication between the various Armadillo modules. The Ports abstract the communication process between modules, and thus prevent modules from knowing details about their neighboring modules. For example, the CPU knows only that the module connected to its `to_mem` and `from_mem` Ports will accept its memory requests and provide replies when appropriate. The actual module could be a memory, a cache, or even a multiprocessor shared memory module. This abstraction allows the simulator to be reconfigured with minor or no changes to the source code (all changes are in the small configurator-generated code files).

In addition to supporting communication between the reconfigurable modules, the Port communication abstraction also allows the virtual modules to be simulated on different host processors, with the Port messages sent over a network between the host processors. This will provide more flexibility in a future distributed version of Armadillo in partitioning a multiprocessor simulation on several host processors, since the partitioning can be done at the Port or module level, rather than only at the node level.

The next section provides details about the use of events within the simulator. Section 5.2 discusses the operations performed during an instruction's processing in the CPU. Next, Section 5.3 presents details on how the simulator accurately handles register renaming, data dependence analysis, and resource scheduling, and other aspects of the processor simulation. Section 5.4 discusses the memory hierarchy and its modules. Lastly, Section 5.5 provides information about the Port objects.

5.1 Events

An Event class was constructed as a C++ template class. An Event is able to remember a time at which the event should occur, an object, a member function of that object, and any parameters to that function. For example, one can create a decode event for a processor with code similar to:

`MakeEvent(now+10, processor1, DecodeNext, argument1, argument2);`

Here, the time is specified as 10 cycles in the future, at which time the function `processor1.DecodeNext (argument1, argument2)` should be called.

A similar C-style mechanism that used function pointers and allocated copies of the arguments would work for some situations, but such a mechanism can not handle member functions of C++ objects.

One disadvantage to the Event class is that the creation and execution of an event both require virtual function lookups, which are much more expensive than ordinary function calls. Thus, Events are an important but costly resource whose use should be minimized as much as possible without sacrificing simulator accuracy or code simplicity.

5.2 Life-Cycle of an Instruction

An instruction goes through several distinct phases in our virtual processor. A diagram of the CPU, for reference, is presented in Figure 3 in the next section.

The first such phase is the fetch phase. An instruction is fetched by the processor from the read-only instruction memory. To improve the simulator's performance, the instruction-fetch memory accesses are not simulated – the instruction is fetched instantaneously from what could be considered a perfect, "warm" I-cache resident on the processor. We are currently not interested in studying the effects of instruction fetching on other aspects of the processor. Of course, Armadillo could be modified to fetch instructions through the memory hierarchy if desired.

Once an instruction has been fetched, its opcode is decoded to decide what type of instruction it is, and which registers it uses. In particular, a distinction is made between registers that are read (a source operand) and registers that are written (a destination operand).

After the decode phase, the data dependence analysis and register fetches are performed. For each source operand to the instruction (this includes any special register values, such as the condition codes), the processor looks at the appropriate register in the register file. If the register has a valid value, it is copied to the source operand. This provides the instruction with a private copy of the register, and thus the register is free to be overwritten without affecting any instructions that require its previous value. A register holds an invalid value when a previous instruction that writes to that register has not yet completed. In this case, a pointer to the Instruction (the "producer") that is going to write a value is stored in a field of the Register data structure. The instruction currently being decoded is added to the producer's list of "consumer" Instructions.

After all the source operands are inspected, the processor looks at the registers that will be written by this instruction. Again, note that any condition code registers may be implicitly updated by the instruction, and need to be handled in addition to any of the general-purpose registers. For each destination operand, the processor modifies the register's fields in the register file to denote that the register file does not contain a valid value for this register, and that this instruction will be the producer for the register value. The destination operand modifications must be performed after the source operand lookups – otherwise, for example, an increment instruction, `addi r1,r1,1` would mark itself as the producer for register 1, and the subsequent source operand lookup would add the `addi` instruction to its own consumer list. The instruction would stall forever, waiting for it to produce a value for itself.

The above actions, taken on behalf of the source and destination operands, implement both register renaming and data dependence analysis. Further details and an example will be given later in this chapter in Section 5.3.1.

Once all source operands are examined and treated appropriately, and the register file has been modified to reflect the destination operands in this instruction, the processor looks at how many source operands are still waiting for register writes from earlier producer instructions. If all source operands are ready, then the instruction is ready for dispatch to a functional unit. Otherwise, the instruction is moved into the decoded instruction pool, where it will remain until all of its source operands have values. Note that the instruction pool is unordered. Several early superscalar processors (PowerPC 601 [15] and Pentium [11], for example) did not have such unordered instruction pools. The PowerPC 601, for example, may execute a floating point instruction out-of-order with respect to an integer instruction, but all floating point instructions are executed in-order with respect to each other, and all integer instructions are executed in-order with respect to each other. That is, the PowerPC 601 has a FIFO queue for each pipeline unit. This limits the instruction reordering to only instructions that do not use the same functional unit. Our virtual CPU and recent superscalar processors, such as the Pentium Pro [12], and the MIPS R10000 [14], allow reordering of all instructions, with only data dependences as restrictions.

When an instruction is ready for dispatch, it is sent to the incoming queue of the appropriate functional unit[1]. If the functional unit has multiple pipelines (only the integer unit can have multiple pipelines), the instruction is dispatched to the pipeline with the shortest queue. The Armadillo simulator does not have a limit on the number of instructions that can be queued at each pipeline, although this number is of course limited by the total number of active instructions, which can be limited by Armadillo. As noted earlier, this lack of many hard-coded limits in Armadillo prevents it from accurately modelling a real processor architecture, but it also allows Armadillo's virtual processor to operate to a large extent in a dataflow fashion, with execution limited only by true data and resource dependences.

Since all instructions dispatched to a functional unit have all of their source operands available, no stalls will occur in the pipeline. This also means that no forwarding to instructions in other pipelines needs to be done – all consumers of a destination operand are still waiting in the decoded instruction pool.

[1]In this paper, a processor has at most one functional unit of each type, but may have multiple pipelines within a functional unit

Once an instruction finishes its pipeline, it begins its completion phase. It forwards all of its destination operands to the appropriate consumers. In addition, the Instruction checks the register file to see if it is still the producer for its output registers. If not, it means that another destination operand that writes to the same register (a Write-After-Write, or WAW, hazard) has been decoded. In this case, no update of the register file should be done – all instructions that needed its value were put on its consumer list, and have already had the correct value forwarded to them.

Although speculative execution is not yet supported, many mechanisms to ease its implementation are already in place. Here, the completion stage of the processor would need to be modified such that once a speculative instruction was ready to be completed, it was placed in a completion queue, whose contents are later handled or flushed, depending on whether or not the speculative branch was correctly chosen. Of course, nonspeculative instructions can be completed immediately.

5.3 Implementation of the Virtual CPU Object

The virtual CPU is a superscalar, pipelined, PowerPC-variant processor. The CPU is presented in Figure 3. As mentioned above, the instruction memory is modelled as a perfect I-cache. This is implemented by allowing the virtual processor's decode unit to have direct access to the text segment of the simulated program, rather than having it perform fetches through the memory hierarchy. The decoded instruction pool keeps instructions that have been decoded but can not yet be dispatched due to a data dependence (only RAW dependences prevent dispatching – WAW and WAR dependences are removed via register renaming). Instructions in the pool receive register values from the register file, or directly from the producing instruction when it completes. Once an instruction is ready to be dispatched, it is sent to the appropriate functional unit. Lastly, the load/store functional unit has a connection to the CPU's ports to the memory hierarchy.

Several different constructs are used to represent various aspects of the CPU internally. Most of these correspond directly to a C++ class of the same name. A brief description of most of these classes is given below. Further elaboration will be provided as needed later on.

CPU: The entire processor (excluding the memory hierarchy).

Universe: One consistent possible path of execution, including a register file. (With speculative execution, two separate execution paths may be explored, leading to two "parallel universes," one of which will be annihilated once the branch condition has been decided. Currently, branches stall further decoding, so only one Universe exists in a CPU.)

FunctionalUnit: The base class for all functional units. A single functional unit (such as the Integer Functional Unit) may have multiple pipelines within it.

Instruction: All information pertaining to the execution of one instruction.

OpcodeInfo: An attribute of an Instruction, containing static (unchanging) information about an instruction, such as the register indices to use, and the FunctionalUnit that the instruction uses.

Operand: A register value (an integer) to be used by the instruction. For example, if the instruction were `add r0, r0, r1` (`r0 += r1`), there would be three Operands – two source (r0 and r1) and a destination (r0). This is part of the dynamic information for an Instruction – although the register numbers are fixed, the actual values added together are determined dynamically.

Port: A one-way communication channel from one module to another. Most modules have an input Port and an output Port for each neighboring module, in order to provide two-way communication.

5.3.1 Instruction Fetch and Decode

When an instruction is decoded (which turns 32 bits of machine code into an Instruction object), certain information is derived purely from the actual binary instruction (*i.e.,* the major and minor opcodes), while some other information depends on the state of the processor (such as the register file's contents). Since some information is static, and thus could be "cached" by the simulator, an Instruction object has a data member called OpcodeInfo to hold this data. Although our simulator does not cache OpcodeInfo data, it would be a simple modification to do so, with potential performance improvements.

The OpcodeInfo data structure holds several useful pieces of information about a 32-bit encoding for an instruction. First of all, it holds a decoded form of the instruction. This allows easy lookup of various bitfields of the instruction. It also holds any special flags for the instruction, such as a bit to specify whether or not it is a memory reference instruction.

Lastly, when debugging is enabled, the OpcodeInfo also holds a short string representing the mnemonic for the major and minor opcode. This makes low-level debugging of the simulator much easier.

An Operand is a private register value that is used by Instructions. There are two kinds of Operands: SourceOperand, which is read-only, and DestinationOperand, which is write-once-read-never. The DestinationOperand is used for modifications to registers, and thus contains a pointer to the Register in the register file to which its value should be eventually stored. It also keeps a list of "consumer" SourceOperands, which corresponds to the Operands in other decoded instructions that are waiting in the decoded instruction pool for forwarding of the computed value.

The DestinationOperand will be explained first. When a value has been computed, and the instruction is ready to store it into the register file or forward it to other instructions waiting for the result, the simulator stores the value into a DestinationOperand. This operation has the side effect of storing the value into the correct

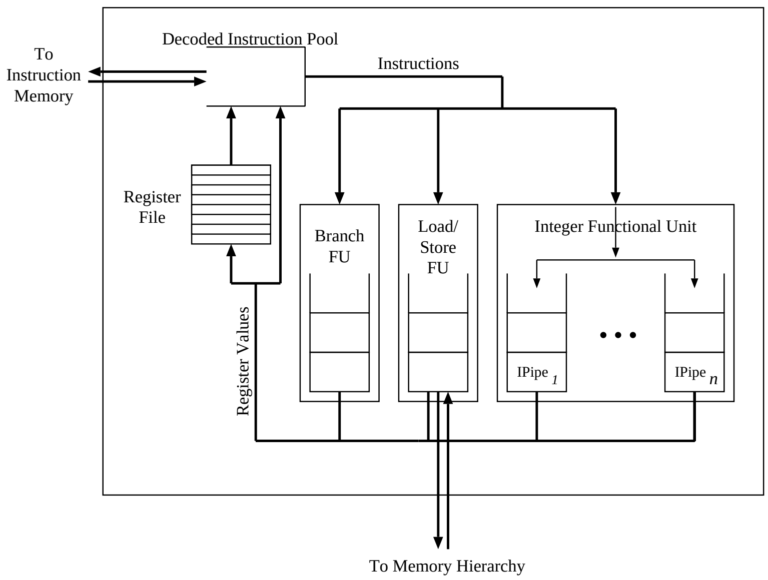

Figure 3: Diagram of the Virtual CPU

Register in the register file, if appropriate, and also forwards the value to any Instructions that are waiting for the result. By making the DestinationOperand write-once and read-never, it keeps the code for the execution of an instruction cleaner and less likely to develop bugs (such as multiple writes to the same DestinationOperand, which could lead to some consumers using one value, and others using the other value).

Note that if a DestinationOperand refers to Register n, and then a later instruction is decoded that should have one of its results also stored to Register n (that is, a Write-After-Write (WAW) hazard), then the later Instruction performs several actions when it is decoded:

- Notify the previous producing DestinationOperand that it is no longer responsible for writing its result back to the register file.
- Notify the Register in the register file that all future attempts to use its value should be forwarded to this DestinationOperand (so that the request can be queued onto the DestinationOperand's consumer list).
- Set its internal data structures so that it knows to write its result back to the register file. Of course, this may be changed by a later instruction.

The SourceOperand is actually no more complicated than a DestinationOperand – it basically performs the dual of the actions of a DestinationOperand. When an instruction is decoded, if the appropriate register for a SourceOperand is available, its value is copied to the SourceOperand, and is thus ready for use by the FunctionalUnit. Otherwise, the Register must be waiting for a previous instruction to complete. If so, the Register has a pointer to the appropriate DestinationOperand that will provide the result. The SourceOperand then asks this DestinationOperand to add the SourceOperand to the end of the DestinationOperand's "consumer" list. When the DestinationOperand is written, it then notifies each SourceOperand on its consumer list of its value, as explained above.

An Instruction has an array of SourceOperands. This allows an Instruction to keep a "private" snapshot of pertinent register values, thus allowing the value in the register file to be overwritten as early as possible.

These techniques in general provide the basic support for register renaming. The missing piece is how a Register knows whether or not it has the current value. Each Register has three data fields: a value (for when the register holds an appropriate value), a pointer to a DestinationOperand that is going to produce its value (when it doesn't have an appropriate value), and a flag to tell which of these states, valid or invalid, the register is presently in. These values are set appropriately during decoding, when a Register's producer field may be set, and the flag changed to invalid, and during completion, when a Register's value field is set, and the flag is changed to valid.

An example of the register renaming and data dependence analysis is illustrated below. The code corresponding to this example is:

```
0x1000: sub  r0, r4, r2   # r0 = r4 - r2
0x1004: stw  r0, 0(r6)    # M(r6+0) = r0
0x1008: and  r0, r0, r4   # r0 = r0 AND r4
0x100C: add  r3, r0, r4   # r0 = r6 + r4
```

Note that this code contains all three types of data dependences: Read-After-Write (RAW), Write-After-Read (WAR), and Write-After-Write (WAW). The `and` instruction contains all three hazards for r_0 (RAW with the `sub`, WAW with the `sub`, and WAR with the `stw`). With register renaming, the destination register for the `and` instruction could be changed to a rename register that was unused (r_7, for example), which eliminates all dependences except RAW. However, our simulator does not explicitly perform register renaming. Instead, as explained above, the simulator keeps "old" values of the register in the SourceOperands of the relevant instructions. In this example, all instructions that require the value of r_0 generated by the `sub` (namely, the `stw` and `and` instructions) would have the result forwarded directly to them, regardless of whether or not the `and` instruction had been completed, so the `and` is free to write back to r_0 as soon as it completes. A detailed, step-by-step example of the dependence analysis and register renaming for this code is shown in the figures below.

Before the instruction at `0x1000` is decoded, assume that the register file has all valid values. This will simplify the diagram. Also, since only register r_0 has data dependences, only its dependence analysis will be described.

In Figure 4, no new instructions have been decoded, and r_0 holds a valid value of 0.

Then, the `sub` instruction is decoded. Register r_0 is marked as invalid, and will be forwarded by a DestinationOperand, D_1, as shown in Figure 5.

The `stw` is decoded, and the r_0 SourceOperand S_1 is added to D_1's consumer list, as shown in Figure 6.

For the next step (the decoding of the `and`), the operation is split into two steps for clarity. First, the source operands are handled in Figure 7: the dependence for the `and` instruction source operand (S_2) is constructed.

Next, the destination operand D_2 for r_0 for the `and` instruction is added to the diagram, as shown in Figure 8. Note that, although the register file no longer contains a pointer to D_1, the `sub` instruction (which is not shown here) still has its pointer to it, so the objects have not been lost.

In the next figure, Figure 9, the `add` instruction has been decoded, and S_3 has been added to D_2's consumer list, as appropriate.

In Figure 10, the `sub` completes, and forwards its value of, say, `0x00000035`, to the consumers S_1 and S_2.

The destination operand D_1 has finished its work, so it is deleted, as shown in Figure 11. The `stw` and `and` instructions can now execute.

In Figure 12, the `and` instruction completes, and forwards its value of `0x00000005` from its destination operand D_1 to S_3 for the `add` instruction, and also stores it into r_0.

Finally, the `add` completes, and all data dependence analysis and register renaming for r_0 is done, as shown in Figure 13.

5.3.2 Register File

The simulated register file has 37 registers. Each register is 32 bits wide. The first 32 registers correspond to the 32 general-purpose registers in the

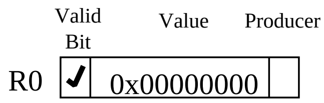

Figure 4: Register-Renaming/Data-Dependence Analysis Example

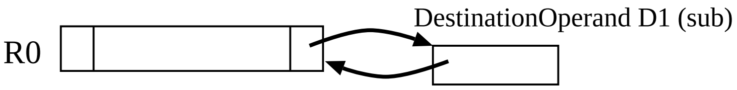

Figure 5: Example (continued)

PowerPC UISA. In addition, the UISA also specifies a program-counter register, a link-register (which holds a single return-address for subroutines), a condition code register, an integer functional unit status register, and a counter register.

The condition code register consists of 8 4-bit fields. Each 4-bit field contains a negative, positive, zero, and summary-overflow bit. The summary overflow bit is set when an overflow occurs, but is only cleared through a specific instruction. All condition-code results from integer operations can be implicitly provided to CR0 (field 0), and floating-point results can be implicitly stored to CR1. In addition, all of these integer and floating-point instructions also have no-condition-code-update versions. There are also CR-manipulating instructions, and compare instructions can store their results to any one of the 8 CR's. For dependence analysis, each CR could be considered a separate 4-bit register (since the values stored to CRi will not affect those stored to CRj). However, for simplicity, the entire condition-code register is treated as a single 32-bit register. Since compiler-generated and hand-written PowerPC code only set condition codes when they are necessary, false dependences on the CR between two instructions are rare.

The integer status register (Integer Exception Register, or XER) has 3 fields implemented in Armadillo. The carry bit is set or cleared appropriately by integer instructions. The overflow bit is set or cleared only if specified by the integer instruction (most integer instructions have a form that sets the overflow flag like `addo` (add with overflow-flag update), and one that ignores overflow, such as `add`). The summary overflow bit is set whenever an instruction sets the overflow bit, and is cleared only through an explicit register-to-register move to the XER. The summary overflow flag is useful when one performs a long sequence of integer calculations and is concerned about overflow. Rather than checking for overflow after each instruction by adding a branch instruction, one can simply test for summary overflow at the end of the calculation, and branch to error-handling code if the SO bit is set.

The count register (CTR) is provided, but no instructions that use it have yet been implemented. It holds a 32-bit counter that can be decremented and tested during some forms of branch instructions.

5.3.3 Pipelines/Functional Units

The CPU has several FunctionalUnits, each of which may have several pipelines within it. An Instruction can be dispatched to a functional unit's pipeline only when all of its SourceOperands are available.

A full circuit-based simulation would take into account the queueing mechanism at the front of each pipeline, and the calculations performed in

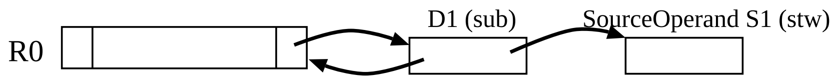

Figure 6: Example (continued)

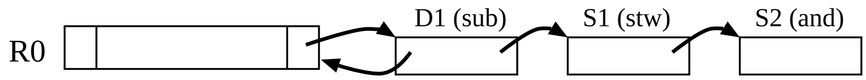

Figure 7: Example (continued)

each stage. However, Armadillo can take several shortcuts to improve performance and still provide an accurate architectural-level simulation.

For example, each stage of the pipeline does not need to be simulated. Since an Instruction is dispatched only when all of its SourceOperands are available, the pipeline will not stall due to data dependences. In addition, since interrupts are not supported, all calculations run to completion. We can accurately model this pipeline by having the operation complete l cycles after it enters the first stage of the pipeline, where l is the latency or length of the pipeline. Thus, no cycle-by-cycle simulation of each stage in the pipeline is needed.

Also, if three Instructions are able to be dispatched to a specific pipeline in the same cycle, no queueing at the head of the pipeline needs to be explicitly simulated. The first Instruction will complete after l cycles, the next after $l+1$ cycles, and the last after $l+2$ cycles. Any later Instructions dispatched to this pipeline will be delayed if they arrive before the previous Instruction has managed to "enter" the pipeline. These shortcuts allow the queueing and execution of most Instructions to require only one scheduled event, the completion event.

Currently, the integer pipeline is unified – that is, each integer pipeline can add, subtract, shift, rotate, multiply, divide, and perform logical operations. It could be split into limited-purpose pipelines (such as a multiply/divide pipeline, a logical/shift pipeline, and an add/subtract pipeline, for example), if desired, for greater simulation accuracy.

5.3.4 Interrupts

Interrupts are not required by the planned experiments, so the CPU does not provide interrupt support. In addition, all code is assumed to be safe and debugged, so bus error and other exceptions will not occur. If any conditions occur that would ordinarily cause an exception, the simulator prints an appropriate error message and terminates the simulation. Although these limitations prevent our CPU from accurately modelling a real processor, the simulator is sufficient for running many types of experiments.

5.3.5 Out-of-order Completion of Instructions

The dispatch mechanisms above (Registers, producer/consumer lists, Operands) detect only true data dependences, and thus restrict the dispatch order as little as possible while still preserving sequential semantics. Since precise interrupts are not supported, Instructions can complete and forward their values as soon as they exit their respective pipelines. This will frequently result in out-of-order completion. Of course, all data dependences are maintained, so program correctness is not compromised.

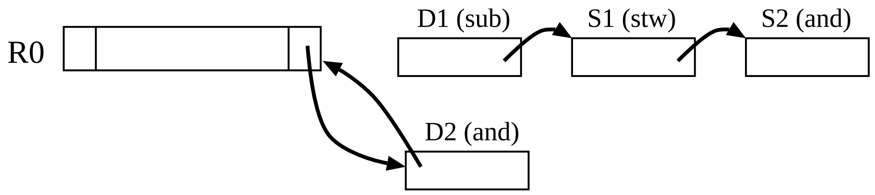

Figure 8: Example (continued)

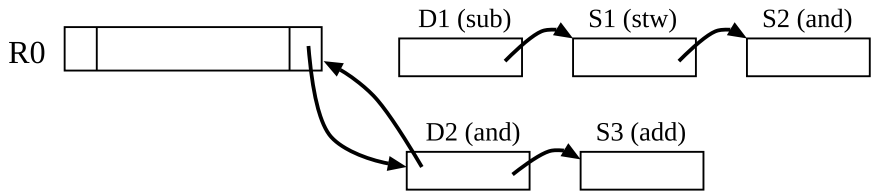

Figure 9: Example (continued)

5.3.6 Universes

A Universe is intended to contain all information needed to allow speculative execution to be attempted along multiple paths. Each possible "thread" of execution would have its own Universe. A Universe spawns a new Universes when a branch instruction is encountered in the instruction stream. The current Universe stops decoding, and the "child" Universe initiates decoding down the predicted thread of execution. If the branch is found to be correctly predicted, the speculative Universe is marked as correct. Otherwise, the Universe is annihilated, and decoding begins on the correct path of execution.

Speculative execution is not supported yet, but with the Universe class already in place, the amount of work required to implement it should be fairly small.

To allow speculative execution, each Universe needs to have a private copy of the reg file, which includes the program counter (PC). Also, each Universe has a list of the instructions that have been decoded but have not yet completed. This list is used for debugging purposes, and the size of this list is used to limit the number of active instructions (instructions that have been decoded but have not yet completed) in the CPU.

Notice that the virtual processor can be modified to be a multithreaded processor with relative ease. A multithreaded processor is one that has multiple program counters (typically between 2 and 8), and tries to decode and dispatch instructions from all active "threads" in a single cycle. Each thread would require its own Universe to handle decoding and to keep its program counter separate from the others. The Universes could either have their own private register file, or could share a common register file.

In addition, each Universe, which is responsible for initiating decoding, has a pointer to a read-only instruction memory. This gives the CPU a perfect read-ahead instruction cache, which improves performance of the simulator by avoiding unnecessary cache and memory accesses.

5.3.7 Ports

The CPU has two Ports (one for input, and one for output, making a bidirectional channel), to the memory hierarchy. Since all network communication is done through the memory hierarchy,

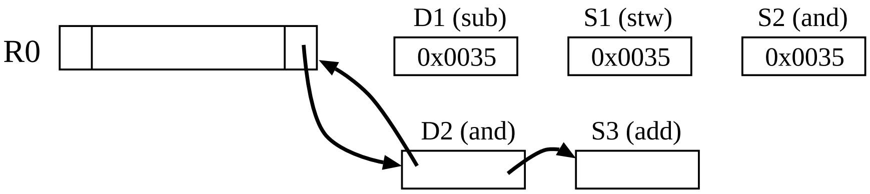

Figure 10: Example (continued)

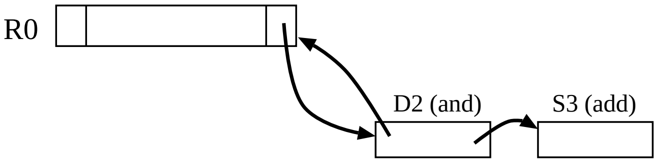

Figure 11: Example (continued)

and the fact that this virtual architecture has no separate I/O bus, the CPU needs no other Ports.

5.3.8 Memory-Ordering Aspect of Load/Store Pipeline

The load-store pipeline does full dependence analysis of all memory operations to allow as many operations to be done in parallel as possible. In addition, memory reference instructions whose effective addresses are different may be reordered. The dependence analysis performed on the effective addresses preserves single-program correctness. However, in a multiprocessor, only weak consistency [7] can be provided. Although a synchronization instruction has not yet been designed, for dependence analysis it would appear as a write with an unknown effective address (such a write must wait until all preceding memory reference instructions have completed, and all later memory reference instructions can not be reordered past it – exactly the desired semantics for a synchronization instruction).

The memory ordering actions are started when a memory operation instruction is first decoded. The instruction is put into the memory-ordering queue, in order to follow proper time-ordering of the memory operations. The dependence graph is updated to include this latest instruction.

The effective-address dependence analysis performed in the load/store functional unit is in addition to the register data dependence performed by the decode unit. Two instructions that do not have data dependences may have memory-ordering constraints, such as a write to an address and a subsequent read from that address. Clearly, the read can not be performed before the write without changing the behavior of the program. On the other hand, two instructions that do have a data dependence (perhaps removed through register renaming) may have no memory-ordering data dependence, such as two reads that load memory values into the same register.

Memory reference reordering, together with the use of split transactions, can greatly improve the performance of memory operations. For example, if we have the following code, with a write miss, the second read can be reordered to be processed before the first read.

0x2000: stw r_1, 0x6000
0x2004: lwz r_2, 0x6000
0x2008: lwz r_3, 0x8000

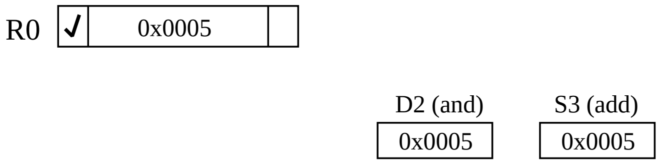

Figure 12: Example (continued)

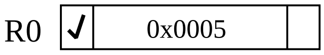

Figure 13: Example (continued)

This reordering prevents the second read from being delayed by the miss penalty of the earlier write and read.

The effective address of the memory reference is unknown until all of the SourceOperands involved in the effective-address calculation are known. In order to preserve correctness, a read with an unknown effective address is forced to depend on all previous writes, and a write with an unknown effective address depends on all previous memory references. Similarly, subsequent reads do not depend on previous reads, including those with unknown effective addresses, and subsequent writes depend on previous writes with unknown effective addresses. Of course, data dependences between operations whose effective addresses are known are also maintained. Table 1 summarizes these dependences. Note that unlike the data dependence analysis for registers, WAR and WAW dependences can not be removed, since the technique of register renaming can not be used for memory.

After each SourceOperand is fetched or forwarded by a completing instruction, the effective address calculation is attempted. If it fails, the effective address remains unknown. If the effective address calculation succeeds, then the load/store pipeline is able to generate new dependences for this instruction, and drop any dependences that are no longer valid. Note that the effective address calculation may be performed before all SourceOperands for the instruction are available. For example, the effective address for $\texttt{stw r}_0\texttt{, 0(r}_{31}\texttt{)}$ can be calculated as soon as the value of register 31 is known.

Once all of the SourceOperands are available, the instruction would ordinarily be ready to dispatch. However, if the dependence graph indicates that this instruction still depends on another instruction (perhaps a read that is unfinished, or even an operation whose SourceOperands are not yet all available), the instruction can not be dispatched.

When a memory operation instruction completes, the dependence graph is again updated to remove dependences on it. Any instructions that were ready to dispatch except for a dependence on this latest completed instruction can now be safely dispatched.

5.4 Memory hierarchy

The memory hierarchy will be discussed from the lowest level up, since the higher levels involve more complexity. The various modules in the memory hierarchy, from top to bottom, are PBus, Cache, MBus, and Memory.

All modules in the memory hierarchy use a memory request, represented by the class MemReq, to communicate. The types of MemReqs that are handled include `READ_WORD`, `READ_REPLY`,

	Earlier Reads with unknown EA	Earlier Writes with unknown EA	Earlier Reads with known EA	Earlier Writes with known EA
Read with unknown EA	no	yes	no	yes
Write with unknown EA	yes	yes	yes	yes
Read with known EA	no	yes	no	if EAs are the same
Write with known EA	yes	yes	if EAs are the same	if EAs are the same

Table 1: Memory Access Dependences
The left column specifies the current instruction of interest, while the top column lists earlier instructions it may depend upon. The table specifies whether or not the current instruction (specified in the leftmost column) depends upon an earlier instruction (specified in the top row).

`WRITE_WORD`, `WRITE_HALFWORD`, `WRITE_BYTE`, `READ_LINE`, `WRITE_LINE`, `INVAL_EA`, and several others. The CPU uses the first five, and other members of the memory hierarchy use those and other messages to implement coherence protocols.

5.4.1 Memory

The Memory module corresponds to one bank of memory chips. It has a specified response time for read requests, and a latency before write requests are complete. Most configurations would use a single bank of memory, although it is possible to set up a multi-bank memory. The Memory may be directly connected to the CPU, or may have other modules in between. The current version of the memory module can handle only one request at a time, and does not support any pipelining (if a read request that takes a 10-cycle latency penalty is going on, no further requests are handled until all 10 cycles have passed). Future versions could provide a write-buffer, and perhaps multiple read ports.

5.4.2 MBus

The MBus provides a bus interface between one or more banks of Memory and the next higher level (*e.g.,* a cache) in the memory hierarchy. Multiple higher-level modules may be connected to the MBus, such as in the case of a cluster of processors sharing one memory. Modules connected to the upper half of the MBus must be able to support coherence MemReqs, such as `INVAL_EA`. Modules connected to the lower half of the MBus are treated by the MBus as memory or memory-mapped modules, that never initiate bus transactions. Memory, L2 Caches, and memory-mapped NIFs, are all acceptable for connection to the lower half of the MBus.

The MBus does round-robin bus arbitration, and supports split transactions. Various Port queries are supported and required (see Section 5.5 below) for proper operation of the bus.

5.4.3 Cache

The Cache is highly configurable. The latencies, line sizes, and associativity can all be modified. The Cache has two sets of Ports: one for connection to a lower module in the memory hierarchy, and one for connection to a higher module. Any appropriate module can be connected at these Ports. The Cache uses a three-state protocol (clean, dirty, invalid) internally, and preliminary support for cache-to-cache coherence policies exist.

5.4.4 PBus

The PBus is very similar to the MBus, except for its position in the hierarchy. The PBus is intended to be more tightly coupled to the CPU. By allowing a PBus between the CPU and the Cache, multiple CPUs could share a single Cache. Also, the NIF may be positioned on the PBus in order to decrease NIF-CPU latencies. Ideally, the PBus clock speed would be faster than the MBus clock speed, and the PBus would also have a much wider datapath, since it is presumably all on-chip with the processor.

5.5 Ports

As mentioned above, all inter-module communication is done through a C++ object for one-way communication called a Port. Most modules use a pair of Ports, to provide two-way communication.

The Port class was created to make it easier to both configure Armadillo and to port Armadillo to run on multiple processors. Since all inter-module communication is done through Ports, a simulator simulating a Port on one processor could theoretically communicate over a network to another processor that is simulating the appropriate module at the other end of the Port.

Besides providing a channel for communication, a Port is also capable of answering several queries concerning the destination end of the Port. The first such query is the `ReadyAt` query. This is used to do the handshaking/flow control over the port. When the source end of a Port asks the ReadyAt query, the Port asks the module at the destination for the clock time at which it will be able to handle a new message. The destination module can return the current time, meaning transmission can begin immediately, or it can return a future time, in which case the source end would typically schedule an event to try again at that time. There is no guarantee that the device will be ready even at that time – the ReadyAt query returns a hint, rather than a promise.

The second and third queries (`WillReply` and `Backoff`) are needed when a bus object needs to ask several modules to figure out which module should respond. (Note that this is not the same as bus arbitration – that is handled separately.) For example, assume we have multiple processors with caches, that are connected to a single main memory on an MBus. If one processor tries to access a memory location that is held in another processor's cache and is dirty, we obviously need to get the data from the correct cache. As another example, if the cached copy of some data is clean, we could retrieve the data from either the cache or main memory. However, since memory has a higher latency (and perhaps lower bandwidth) than a cache, we would rather have the cache respond. The second query, `WillReply`, handles figuring out who is willing to reply for a given effective address or other message parameter. In both of the examples above, both the Cache and main memory would typically respond affirmatively (in the first case, the memory does not know it has an out-of-date value). However, the MBus would know to ask memory last, since it's the last resort for any transaction, so the Cache would win. The WillReply queries can be done in parallel at the same time, with the results priority-encoded to determine the module which will respond.

The third query, `Backoff`, is used to make sure that no one on the bus has an objection to a peer on the bus handling some request for the specified effective address. It is possible for a Cache to have a pending transaction or split transaction for a specified effective address in progress when another Cache tries to do another operation on the same address (or an address within the same cache line). This could lead to inconsistencies. Rather than locking the bus in some way, we chose this backoff message. Once a replier has been chosen (via the WillReply query), the bus asks every bus object whether or not it has any objections to allowing the transaction to be handled by the other bus object. If any module answers affirmatively (a positive number is used, to specify an expected number of cycles that the requestor should wait before retrying), the bus transaction is not allowed to take place.

6 Other Armadillo Features

6.1 Trace Generation

Armadillo currently has the ability to create simple tracefiles. At runtime, the flag `--tracefile=foo` will store a trace of the instructions in decoded order to the file "foo". A sample trace file is shown in Figure 14. The tracefile lists the instruction address, the 32-bit opcode, and the mnemonic for that instruction. This tracing mechanism is implemented with a simple `fprintf` statement in the decode unit, and can be easily modified to provide more information, such as the current simulation time.

Code can be easily added to do more elaborate trace generation (memory-access only with effective-addresses, register-file access, network operations, etc.).

6.2 Debugging Support

Armadillo can be compiled to provide different levels of debugging support for the simulator itself:

- Lightweight sanity checking/assertions.
- More complete sanity checking/assertions.
- Memory allocation and deallocation checking.
- Several levels of internal debugging logging.

Since debugging support typically impairs the simulator's performance, the level of desired support is customizable at compile time and, to a lesser extent, at run time.

Certain assertions are always tested, to provide some amount of protection from programming errors or memory corruption. If the simulator was compiled with the `DEBUG` flag, further sanity checking can be enabled on the simulator command-line using `--sane`. An example of this more-detailed but expensive sanity checking is verifying that every pointer in a doubly-linked list is correct.

In addition, memory-use tracing can be enabled. A user-level `malloc()` package was modified slightly, and the C++ operators `new` and `delete` were redefined to use these allocation routines. When full memory-tracing is enabled, each allocated chunk of memory contains, in its header, a structure that describes where the `malloc()` or `new` call originated in the source code. In addition, the headers are put on a linked list, and are removed only when the pointer for that header is freed. If unfreed memory exists when the simulator tries to exit, the memory-tracing routines print out the list of headers for the unfreed data, allowing the user to quickly find and correct memory leaks.

With the `--debug` runtime flag, the simulator provides verbose logging information to a file (the filename can be selected via the `--debugfile=foo` directive). This logging includes information on the contents of various queues of the simulator, headers of messages between modules, and detail on the contents of internal variables and queues.

The format of a logged statement is as follows:

$\langle$Ln (Logging depth)$\rangle$ $\langle$Simulator time$\rangle$ $\langle$Source file and line number$\rangle$ $\langle n$ blanks$\rangle$ $\langle$Message$\rangle$

A sample section of a log file is presented in Figure 15.

The blanks before the message provide an indentation to the messages, making the hierarchy of loops or procedure calls clearer.

```
0x1c18:  0x38000064 (addi)
0x1c1c:  0x98090000 (stb)
0x1c20:  0x387f0038 (addi)
0x1c24:  0x4bfffd2d (bl)
0x1950:  0x7c0802a6 (mfspr)
0x1954:  0x93e1fffc (stw)
0x1958:  0x90010008 (stw)
```

Figure 14: Sample Trace

```
L01 0010      CPU/bpu.cc(225): BPU Execution of opcode 480000dd:
L01 0010      CPU/bpu.cc( 79): Doing a 'b' instruction.
L02 0010     CPU/value.h( 18):  Original PC:  00001b18
L02 0010     CPU/value.h( 18):  Destination value is 00001bf4
L02 0010  CPU/operand.cc(105):  Waking up consumers of result:
L03 0010  CPU/operand.cc(108):   Register File consumer R24
L03 0010  CPU/operand.cc(125):   Done waking up consumers.
L02 0010     CPU/value.h( 18):  Saving PC of 00001b1c into LR
```

Figure 15: Sample Excerpt of the Debugging Log File

In most situations, one does not need the highest level of information, so the debugging level can be set. Each procedure that uses the logging subroutines typically increments a counter when it performs a lower-level or nested function (such as a loop or a subroutine call). Then, it decrements the counter after the operation. The logging code outputs this counter at the beginning of each line, allowing the log to have a more structured look. The user can specify on the command line `--dlevel=n`, in which case no message with a logging depth greater than n will be printed to the log file. This allows the user to roughly control the verbosity of the log file.

7 Performance

A few simple benchmarks were used to measure Armadillo's performance. These include the Dhrystone benchmark, and several "toy" programs that strongly exercise a specific portion of the simulator. All performance numbers are the average of 10 benchmark executions.

7.1 Dhrystone benchmark

The Dhrystone benchmark was ported to our simulated one-node architecture with some minor modifications. The modifications were as follows:

- A custom `times()`-style function was written that returns the CPU time used by the simulator. A second function was written that returns the number of simulated clock cycles. These allow the benchmark program to print out performance both of the simulated architecture and of the simulator.
- The number of runs was changed from a user-input value to a compile-time constant. This is due to the fact that scanf() has not yet been ported to the simulated architecture.
- The performance calculations at the end of the benchmark were modified to use only fixed-point arithmetic. Armadillo does not yet handle floating-point arithmetic.

The Armadillo simulator has been compiled and tested successfully on three platforms so far: Pentium (running NetBSD), Sun SPARCserver 20 (running SunOS 5.5), and an Alpha (running DEC OSF1). These three platforms differ in both endianness and in processor word-size (the Alpha is a 64-bit machine).

The Dhrystone benchmark was run on the host machine, and on the simulated architecture, to calculate the slowdown of the simulator in a processor-independent and clock-frequency-independent manner. In addition, different simulated architectures were used for different runs to show the simulation overhead of the virtual architectural differences. For example, the simulator runs slower in the base virtual architecture (the CPU is directly connected to the memory, with no intervening cache or bus) than in the CPU-Cache-Memory architecture, even though the simulator needs to perform extra processing to handle the extra Cache module.

The virtual CPU, in all three virtual architectures, was configured to have 4 integer pipelines within the integer functional unit, and the CPU allowed at most 16 instructions to be active at any time. These choices resulted in poor performance for the memory reference synthetic benchmarks, which is discussed below.

The architectures investigated include: Base (CPU-Memory), Cache with MBus, and Cache, MBus, and NIF. In the latter architecture, the Dhrystone benchmark obviously does not utilize the NIF functionality, so the only performance difference between this architecture and the Cache and MBus architecture will be due to checking the extra bus peer, the NIF, on appropriate MBus operations.

The Armadillo simulator processed the Dhrystone benchmark at a rate of 15,000 instructions per host CPU second on the Pentium 90 test machine.

7.2 Armadillo micro-benchmarks

In addition to the Dhrystone benchmark above, various micro-benchmarks were constructed and simulated. For accuracy, several of the benchmarks were hand-coded in assembly. Each benchmark contains 100,000 instructions corresponding to the operation of interest, plus a few more instructions to provide the looping behavior and the stack setup and exit code. These benchmarks were run on all three virtual architectures, but only on a 90MHz Pentium machine.

The implementation of the loop takes three instructions (an increment, compare, and branch), so the loop overhead is less than 1%. In addition, the number of instructions required to implement `__start`, `exit`, and `main` is less than 20 instructions. Out of 100,320 instructions, then, 100,000 of them are the operation under study. Thus, these benchmarks provide an accurate measure of the performance for a specific class of instructions.

The performance for pure CPU instructions is significantly higher than non-CPU-only instructions for two reasons. First of all, all CPU-internal instructions require at most two events (one for decoding, one for completion), and require **no** communication through Ports. Thus, the overhead from these relatively expensive resources (Events and Port messages) is quite small. Second, the CPU was developed before the other modules, and has been optimized the most.

The first benchmark is a loop of 1000 identical `addi` instructions. Since the instructions all use the same register as a source and destination, each instruction depends on the previous, and thus the instructions are forced by their data dependences to be executed in program order.

The second benchmark is a loop of four 250-instruction increment threads, as above (each thread repeatedly increments a different register), except that the four threads have been interleaved. This implies that there are no dependences between any four neighboring instructions, so the simulator can dispatch four instructions per cycle. The difference between this benchmark's results and the previous one's results show how instruction dependences affect performance.

Note that the second benchmark runs faster than the first. Although this may seem counter-intuitive (the second benchmark requires more

Host platform	Host processor	Simulated Virtual Architecture		
		Base	Base w/ Cache and MBus	Base w/ Cache, MBus, and NIF
Pentium at 90 MHz	124793	12.648 (9866)	14.210 (8782)	14.04 (8888)
SuperSPARC	162048	16.5 (9821)	20.01 (8098)	20.08 (8070)
Alpha 21064 at 175MHz	151937	7.853 (19348)	9.326 (16292)	9.961 (15253)

Table 2: Dhrystones and Slowdown Factor (in parentheses) per Host CPU Second.

Benchmark	Virtual Architecture		
	Base	Base w/ Cache and MBus	Base w/ Cache, MBus, and NIF
Arithmetic	28330.8	27763.5	26370.8
Arithmetic (interleaved)	32028.7	31378.2	29166.3
Loads	6414.5	7809.9	8432.8
Stores	5036.2	5110.1	5275.1
Stores (interleaved)	4605.4	5209.4	5661.5
NIF loads and stores	4834.5	4857.7	5230.6

Table 3: Microbenchmarks on 90MHz Pentium (simulated instructions per host CPU second)

detailed data dependence analysis) the cause is due to the overhead of Events in the first benchmark. In the first benchmark, only one instruction completes each cycle. Thus, we end up decoding only one instruction per cycle, and schedule a decode event for the next cycle. Each instruction, then, requires two events – one for its decode, and one for its completion. For the second benchmark, four instructions are completed each cycle, so four new instructions can be decoded in each cycle. Since the decoder can decode four instructions per cycle, and it is allowed to do so every cycle (because four instructions always complete), it ends up decoding four instructions per decode event. Thus each instruction requires, on average, 1.25 events, rather than 2 events as above.

Armadillo could be modified in several ways to improve the first benchmark's performance. The first is to decrease the Event overhead. This will be done on a continuous basis as the simulator is improved. The second alternative is to change the flow-control mechanism for decoding instructions. Currently, the decoder decodes as many instructions as it can (until the number of active instructions has reached the limit for maximum number of active instructions). Then, as each instruction completes, the decoder is notified that it can decode another instruction. Some hysteresis could be added, so that the decoder would only be awakened when some minimum number (such as 4) of slots were open, rather than every time there is at least one available slot.

In addition, there are several memory-system benchmarks. These include loops of 1000 operations, as follows:

- Cache reads (of the same address)
- Cache writes (of the same address)
- Cache writes (four different addresses, interleaved)
- MBus-level reads and writes for the NIF

For the first two virtual architectures, which do not have a NIF, the NIF accesses are treated as accesses to cacheable memory.

As can be seen from the performance numbers, instructions involving the memory hierarchy are very expensive compared to those that utilize only the CPU. Further analysis showed that most of the overhead comes from the memory-reordering code. For example, if 4 memory reference instructions have been decoded and are in various phases of execution, and a later memory instruction has been decoded, this later memory instruction needs to have its dependences checked against all 4 other operations. This allows the load/store pipeline to dispatch this last operation, even if the previous 4 are waiting for a register value, if there are no dependences. In addition, once the effective address (EA) for the operation is known, its dependences need to be rechecked: a read with an unknown EA depends on a write of a specific EA, but a read with a different EA than the write does not depend on it any longer. These recalculations of dependences are necessary to allow reordering of the memory operations, but provide a large amount of overhead.

Note that the virtual architectures used while running these benchmarks were configured to allow at most 16 active instructions. Thus, in the memory tests, the CPU quickly reaches the state where 16 loads or stores have been decoded. Thus, each new memory reference instruction has its dependences checked against 15 other memory reference instructions. In most real workloads, only some subset of the active instructions would be memory reference instructions, leading to a smaller overhead for memory operation dependence analysis. When the number of active instructions was dropped to 4 instructions, the performance (measured in simulated instructions per CPU second) for these same memory benchmarks nearly doubled.

The load/store pipeline can also be configured to be FIFO, in which case these memory dependence checks are not performed, and performance improves.

8 Conclusion

Our initial goals for this project were to create an object-oriented, efficient uniprocessor simulator, with support for enhancing it for multiprocessor simulation. The object-oriented design was successful in allowing easy reconfiguration of the virtual architecture, as discussed in Chapter 4. It also makes future improvements to the virtual architecture (such as adding speculative execution or modifying the processor to be multithreaded) require less redesign. Our goal for simulation efficiency was to be able to simulate at least 10,000 instructions per second. Although very long strings of load/store instructions fall short of this goal, more realistic workloads (like the Dhrystone benchmark) tend to run above 15,000 instructions per second, significantly surpassing our original goal. In addition, performance improvements for several of the bottlenecks are known, and only require implementation. Lastly, the first steps towards allowing Armadillo to simulate multiple processors have already been taken, and little restructuring of the uniprocessor simulation code has been required, as expected.

References

[1] L. A. Barroso, S. Iman, J. Jeong, K. Öner, and M. Dubois. "RPM: A Rapid Prototyping Engine for Multiprocessor Systems,"

IEEE Computer, Vol. 28, No. 2, pages 26–34, February 1995.

[2] E. Brewer, C. Dellarocas, A. Colbrook, and W. Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, 1991.

[3] M. Brorsson, F. Dahlgren, H. Nilsson, and P. Stenström. "The CacheMire Test Bench – A Flexible and Effective Approach for Simulation of Multiprocessors," In *Proceedings of the 26th Annual Simulation Symposium*, pages 41–49, 1995.

[4] S. Chandra, J. Larus, and A. Rogers. "Where is Time Spent in Message-Passing and Shared-Memory Programs?" In *ACM SIG Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 51–60, 1994.

[5] H. Cragon. "Executable Instruction Set Specification," *Computer Architecture News*, Vol. 11, No. 1, pages 25–43, March 1983.

[6] H. Davis, S. Goldschmidt, and J. Hennessy. Tango: A Multiprocessor Simulation and Tracing System. Technical Report CSL-TR-90-439, Stanford University, 1990.

[7] M. Dubois, C. Scheurich, and F. Briggs. "Memory Access Buffering in Multiprocessors," In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, 1986.

[8] S. Goldschmidt. *Simulation of Multiprocessors: Accuracy and Performance.* Ph.D. dissertation, Stanford University, 1993.

[9] S. Goldschmidt and J. Hennessy. The Accuracy of Trace-driven Simulations of Multiprocessors. Technical Report CSL-TR-92-546, Stanford University, 1992.

[10] M. Hill, J. Larus, A. Lebeck, J. Lewis, and D. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. Technical Report CS-TR-92-1122a, University of Wisconsin-Madison, 1992 (revised January 1993).

[11] Intel Corporation. *Intel Pentium Processor Technical Backgrounder.* Available on-line at `http://www-cs.intel.com/oem_developer/microprocessors/pentium/7183.HTM`.

[12] Intel Corporation. *A Tour of the P6 Microarchitecture.* Available online at `http://www-cs.intel.com/oem_developer/microprocessors/pentium_pro/7164.HTM`.

[13] V. Karamcheti and A. Chien. "Software Overhead in Messaging Layers: Where does the time go?" In *ASPLOS VI (ACM SIG Arch. Support for Prog. Lang. and Operating Sys.)*, pages 61–73, 1994.

[14] MIPS Technologies Inc. *The R10000 Technical Brief.* Available on-line at `http://www.mips.com/r10k/r10000_Pr_Info/Tech_Brief.html`.

[15] Motorola Corporation. *PowerPC 601 RISC Microprocessor User's Manual*, 1993.