WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

DUSTY CACHES TO SAVE MEMORY TRAFFIC

by

Scott J. Friedman B.S. Applied Science

Prepared under the direction of Dr. Ron K. Cytron

A thesis presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Master of Science

May, 2005

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

DUSTY CACHES TO SAVE MEMORY TRAFFIC

by Scott J. Friedman

ADVISOR: Dr. Ron K. Cytron

May, 2005
Saint Louis, Missouri

Reference counting is a garbage-collection technique that maintains a per-object count of the number of pointers to that object. When the count reaches zero, the object must be dead and can be collected. Although it is not an exact method, it is well suited for real-time systems and is widely implemented, sometimes in conjunction with other methods to increase the overall precision. A disadvantage of reference counting is the extra storage traffic that is introduced. In this paper, we describe a new cache write-back policy that can substantially decrease the reference-counting traffic to RAM. We propose a new cache design that remembers the first-fetched value of a cache subblock, so that the subblock need not be written back to RAM unless a different value is present. We present results from experiments that show the effectiveness of this approach, particularly in mitigating the storage traffic due to reference counting.

For My Friends and Family

Contents

List of Figures

Acknowledgments

Thanks to NSF for their funding the Liquid Architecture project with grant ITR–0313203, and to Washington University for granting me sanctuary in their walls for another two years.

A warm thanks to the students and members of the Distributed Object Computing Group for providing me smiles and companionship while I earned my degree. Much of my research is supported and inspired by the members of the Liquid Architecture group, and I deeply appreciate their work and support; it's been a pleasure working on the vanguard of computing technology with such determined individuals.

Special thanks to my advisor of great magnitude, Dr. Ron K. Cytron, who inspires me to cultivate within myself the humor and enthusiasm that he carries with him each day.

Finally, I thank my parents, my sister, and my fiancee, all of whom have supported me in my higher (and higher) education, and express a genuine interest in the welfare of my research. I can sense the turmoil it causes them to try to keep their eyes from glazing over when I discuss the details of software profiling with reconfigurable logic. If any of you are reading this, I give you permission to set it down now, though I deeply appreciate the gesture.

Scott J. Friedman

Washington University in Saint Louis
May, 2005

Chapter 1

Introduction

The relative cost of a RAM access is worsening; that is, the speed of on-board operations and cache memory is increasing faster than the speed of the memory bus and the memory itself. As Hennessy and Patterson point out, memory access speed has gained 7% per-year performance improvement in latency since 1980, and microprocessor performance has improved between 35% and 55% per year since 1980. This discrepancy is inconvenient for programmers and engineers who desire fast performance, not to mention real-time programmers who require predictable, consistent software performance. Hennessey and Patterson agree that "clearly, there is a processor-memory performance gap that computer architects must try to close" [5].

The above suggests that we should perform our operations on-chip as often as possible, opposed to climbing the "memory wall" [15] and succumbing to the lower speeds of buses and memory. Historically, the most common remedies to this problem are adding (more) levels of blazing-fast on-chip cache memory and optimizing cache behavior to absorb more of the main memory traffic.

In this thesis, we examine the effectiveness of a "dusty" write-back cache policy that suppresses unnecessary writes to memory. One scenario in which this design can prevail is reference counting, a garbage-collection technique that is widely implemented and well-suited for real-time systems. Reference counting maintains a count of pointers that reference every object; these pointer-values change rapidly in scenarios of complex pointer arithmetic. In many cases, a reference count changes from its value v, but then quickly returns to v. Normal write-back cache policy marks this reference count *dirty* upon any value change, marking it for eventual copy back to RAM. When the value returns to v, it is still considered dirty, and when it is evicted from cache, it will be unnecessarily written back to memory. We introduce a new

method of determining how "soiled" a value is, and recognize that a value is indeed *dusty* when it is altered in cache, but only dirty when it differs from the value in memory. We examine reference counting with this dusty write-back policy in this thesis.

Implementing a new cache policy is a microarchitecture optimization; such optimizations are traditionally costly to implement in hardware, and are time-consuming to simulate in software. Traditionally, to test our new cache policy, we would have to first spend a long time in software simulation, then make an **Application-Specific Integrated Circuit** (ASIC) that uses our cache design, costing millions of dollars. However, the Liquid Architecture [8] system, developed at Washington University under grant ITR–0313203, allows us to reify and analyze our proposed microarchitecture rapidly and with no additional monetary overhead.

In the body of this thesis, we first discuss background details that are necessary to understand the following experiments. We then propose the new write-back cache policy, statistically examine its effectiveness with reference counting and Java SPEC Benchmarks, and use the Liquid Architecture platform to implement and quantitatively analyze the microarchitecture optimization at clock-cycle resolution. Finally, we propose several avenues of related future work, and review the experimental findings of this thesis.

Chapter 2

Background

In this chapter, we review microarchitecture optimization, cache organization, and cache write policy to help develop the foundation for our further investigation. We also discuss reference counting and some common trends of **Object-Oriented Programming** (OOP), both of which are utilized in the included experiments.

2.1 Architecture vs. Microarchitecture

This thesis proposes a cache microarchitecture optimization, so we must make the distinction between software, architecture, and microarchitecture optimizations. We will take an outside-in approach, shown in Figure 2.1.

- Programmers often optimize their code to make it more efficient. They depend on compilers to accurately translate their code into architecture-specific machine

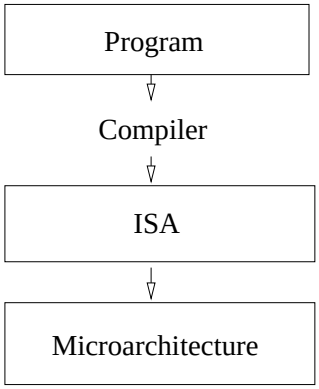


Figure 2.1: The tiered relationship of program, architecture, and microarchitecture

instructions. In many cases, an understanding of microarchitecture concepts such as cache locality and branching behavior can help a programmer with such optimization.

- Compiler developers must provide this translation, but make optimizations of their own to produce a more efficient list of machine instructions, counting on the computer architecture to interpret these instructions as expected.

- Computer architecture developers provide an instruction set architecture (ISA), or an interface for the compiled program to interface with the microarchitecture beneath.

- The *microarchitecture*, or chip, must realize the ISA above it and govern the primary elements of computation. As is proposed in this thesis, we can bias the microarchitecture to take advantage of common programming idioms to make programs run more efficiently. Optimizations at this level must not change computational behavior with respect to the architecture.

Cache memory itself is a microarchitecture optimization, and it lends itself to further behavioral optimization, as we conclude later in Subsection 2.2.1.

2.2 Cache

On-chip cache memory is the first and most effective safeguard against unnecessary memory accesses. It provides a quick buffer layer for intercepting temporally local memory accesses. Cache memory is often on the processor itself, so due to its proximity and access speed, it saves us the long trip to RAM.

Cache Organization

We can first classify cache memory by the nature of the data it buffers. *Instruction cache* (I-cache) contains instructions for processor execution, and for the purposes of this thesis, is read-only. *Data cache* (D-cache) buffers the data-values from our software execution, including reference counts. These caches are often separated on the processor, but may be combined to form a *unified cache*.

Cache memory is composed of *blocks*, also known as *lines*, which are fixed-sized data collections. Often times, multiple values such as integers and characters share

a single block, and blocks are often decomposed into *subblocks*. When a block of main memory is accessed, it is immediately copied to cache, so that any subsequent memory accesses will use the cached copy and avoid the trip to main memory.

For each block, the cache maintains the block addresses and offsets, as well as a *valid bit* for each subblock to monitor whether the subblock contains a valid value. This improves the cache search speed, and will help prevent valid subblocks from being overwritten when invalid subblocks can be written instead.

When the CPU requests to read a value from memory, we first search the cache. If the value is cached, we experience a *read-hit*, and the value is accessed in cache. If, however, the value is absent in cache, we incur a *read-miss* penalty of returning the value from RAM and storing it back in the cache. When the value is stored back in cache, we may have to *evict* a valid subblock from our crowded cache. Several eviction strategies are utilized, such as completely *random* eviction, *last recently used* (LRU), and *first in, first out* eviction, where the oldest block is evicted. Reads compose roughly 79% of all data cache traffic [5], and the eviction rate depends on read locality. The penalty of a miss depends on the latency and bandwith of the memory and bus, but as discussed in the previous chapter, the relative penalty is worsening.

Writes compose 21% of overall memory traffic [5]. Just as with reads, the cache experiences *write-hits* and *write-misses*, but the behavior of each differs depending on which *write-policy* the cache employs. Historically, there are two basic strategies; we explore these below.

Write-Through Cache

The most primative accepted cache policy is *write-through*. In this cache configuration, whenever we write to a memory address, it writes directly to main memory. This results in a memory write every time, and the CPU waits for the write to complete in what is called a *write stall*. To help avoid this stall, the processor can instead write to a *write buffer* and continue its processing, while the write buffer executes the write to memory in parallel. The processor may then quickly write the same value into cache (known as *write allocate*) or not write it into cache, and modify the value in RAM alone via the write buffer (*no-write allocate*).

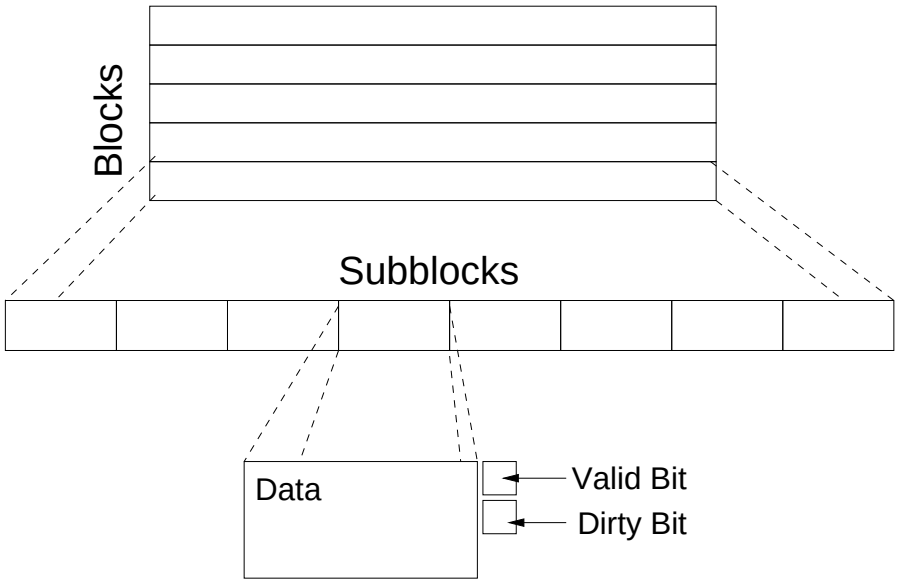


Figure 2.2: Write-back cache organization

Write-Back Cache

A widely accepted alternative to the simple write-through cache policy is *write-back* cache, pictured in Figure 2.2. This configuration has the same benefits as write-through cache for reads, but it can save on write-hits. It achieves this by reflecting all writes directly into cache memory. Unlike write-through, it does not write to main memory at this point. Instead, it keeps a *dirty bit* per subblock in addition to the valid bit that signifies whether the value in cache has been altered since it was read in from the memory. When the value is evicted from cache, it is written back to memory if the dirty bit is set.

2.2.1 Further Cache Optimization

We shall show that a standard write-back cache policy lends itself to some unnecessary memory writes. If a value is altered and promptly returns to the same value in a following write, it is still marked dirty and is written back to main memory even though the value in main memory is identical. Moreover, if we have a value k in memory and we write the very same value k to the same address, the write will cause us to mark the subblock dirty, and k will be unnecessarily written back to memory upon eviction. We can conclude that the dirty bit is sufficient but not necessary to judge whether a value needs to be written back to memory.

We investigate a more effective cache design that writes to memory only if the value it writes back is indeed different. This can save memory accesses, especially for software operations and idioms that involve changing a variable and then promptly returning it to its former value.

2.3 Reference Counting

Reference counting [14] is an efficient albeit inexact means of automatic memory management, otherwise known as garbage-collection [14]. Garbage-collection entails automatically reclaiming heap-allocated objects from memory once they are no longer needed by a program, and is utilized in languages such as Java and C#. Reference counting is one avenue to garbage-collection functionality; it works by counting the pointers that reference each object. When the count reaches zero, the object is "garbage" and may be collected. Reference counting is well suited for real-time systems and is widely accepted and implemented [1].

Though the implementation is straightforward, reference counting can impose considerable overhead due to increased memory traffic. Examples of said traffic are found in common OOP patterns that have one object point to another for a short time, before pointing away. The Iterator pattern [4] is a very simple and frequently deployed example of this *thumbing-through* behavior.

2.3.1 The Thumb Idiom

This widely used *thumb object* programming idiom entails a pointer referencing an object, performing a short set of operations then pointing away, often to another object. We observe this behavior often when iterating through any data structure such as a linked list, tree, vector, or hashtable. This behavior is also common in sorting algorithms. We observe the thumb object nature of the Iterator pattern with respect to reference counting.

Iteration with Reference Counting

A common use of the Iterator pattern is shown below, where we traverse an entire list and process each item in the collection. It seems simple in nature, but it causes a lot of reference counting traffic.

```
LinkedList list;
...
Iterator iter = list.Iterator();
while (iter.hasNext()) {
    Object item = iter.next();
    foo(item);
}
```

As we iterate over the list, the Iterator's internal place-keeping pointer switches from node n_{i-1} to n_i, and onward to n_{i+1}, until the end of the list. The reference count of node n_i increments from k to $k+1$ once the iterator touches it, remains at $k+1$ for a short while, then decrements back to k as the iterator moves onward to node n_{i+1}. It thumbs through every node in such, until termination.

Each node in the list will have a similar reference count "hiccup" due to this common thumb object idiom. We next discuss how the cache behavior differs with respect to the write-policies we designated above.

Cache Response to Iteration & Reference Counting

With a write-through cache policy, both the increment and the decrement will be written directly to memory when we point to and away from the object, respectively.

With a write-back policy, the reference count is marked dirty after the increment, and remains dirty after the decrement. Though the reference count is the same before and after the short *hiccup*, the write-back cache believes it to be dirty. Therefore, upon eviction from cache, the value is unnecessarily written to memory. This will occur every time a thumb-pointer (or any pointer) points to an object, then away again.

With our dusty cache policy, we would experience no writes to memory for such hiccups; just a single read from memory to get the reference count into cache. Upon the value's eviction from cache, the microarchitecture finds it to be identical to its former value and does not write it back to memory. We will discuss this configuration more in Chapter 4.

Chapter 3

The Liquid Architecture System

The Liquid Architecture system takes advantage of reconfigurable logic to permit timely design, prototyping, and analysis of new hardware modules. Without such a tool, the dusty cache idea could not as easily have been prototyped, tested, and analyzed in ample time for me to write this and graduate. Moreover, to synthesize a microarchitecture without reconfigurable logic requires millions of dollars.

Our Liquid Architecture research team recognized these obstacles in the hardware design and software profiling processes, and developed an interactive system to remedy them [11].

3.1 The Profiling Problem

Programmers often want to know how their software utilizes the underlying microarchitecture. With an accurate view of what happens on-chip during a program run, a programmer may optimize his or her software to take better advantage of the hardware beneath. Such software-microarchitecture interaction feedback is surely useful, but very difficult to gather. Unfortunately, many methods of gathering accurate software performance data have fundamental flaws in accuracy and timeliness.

Profiling software performance with other instrumented software can yield skewed results. In some instances, the profiling software will add extra overhead and we're left with a faulty report of processor activity. Other times, software profiling will not provide sufficient resolution of results, and the results are too vague to draw conclusions. Simulation suites, however, yield extremely fine resolution but take an extremely long time to profile the simplest of programs. Moreover, many

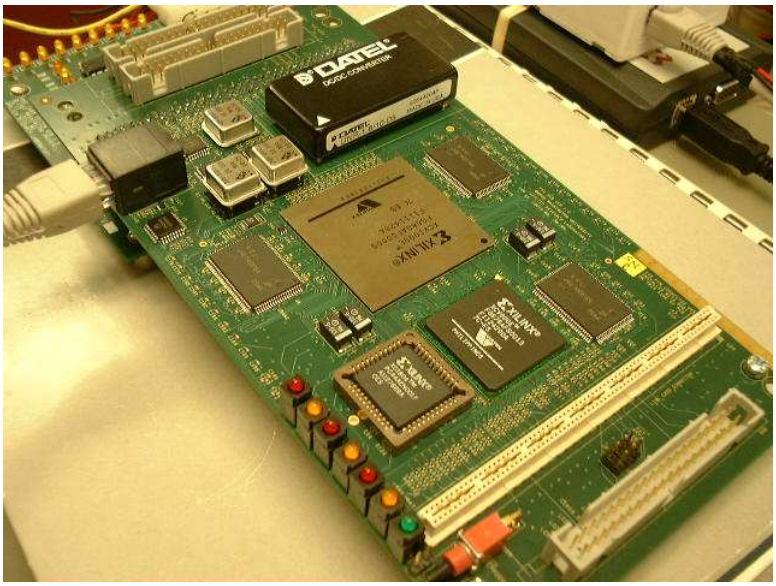


Figure 3.1: Photograph of the FPX

software profilers and simulation suites do not account for (or cannot adequately predict) some of the extremely improbable events that occur during normal execution such as pipeline stalls and store buffers.

3.2 The Liquid Architecture Solution

The Liquid Architecture solution combines reconfigurable logic, microarchitecture support for monitoring on-chip events, and a web-based configuration and analysis interface. This offers a solution to the above profiling problems and permits rapid design and testing of hardware and software structures, making this thesis plausible and conclusive.

3.2.1 The Liquid Processor

The Liquid Architecture processor began as LEON2 [3], a softcore processor for embedded systems, developed by the ESA (European Space Agency). The LEON core provides sophisticated architecture features such as instruction and data caches, the entire SPARC V8 instruction set [7], and buses for high-speed memory access and low-speed peripheral control.

The Statistics Module

The group modified the core to add the *Statistics Module* [6], a performance-measurement functionality for obtaining cycle-accurate timing results, cache-behavior statistics, and method-specific output for each. Such statistics are typically unavailable in generic processors, and are incredibly monotonous and time-consimuing to attain through simulation - the Liquid Architecture processor runs programs at full (FPGA) speed.

This module is really a collection of smaller counter modules, each of which has the following:

- One specific instruction or event to track
- One counter to track how many times this instruction or event has fired
- Two memory addresses (a low and a high, to constitute an address range)
- A connection to the address bus
- A connection to the event bus
- A connection to an output data bus

With this information, each counter can listen on the buses, and if the event occurs within the designated memory range, the counter is incremented. This is all done in parallel, so the tracking mechanisms do not add extra clock overhead to the execution of the program.

The entire module is customizable; that is, we can instantiate varying numbers of these tracking modules within the statistics module with a simple change to the **VHSIC Hardware Description Language** (VHDL) specification. Once instantiated, we can send packets to the microarchitecture to program the instructions and addresses for each counter module of the Statistics Module.

One precaution the Statistics Module takes is overflow prevention. When a user-designated amount of clock cycles expire, the entire module evicts the data from its counters and passes the statistical data to the packetization module to be sent back to the user. It then resets the counters and continues monitoring execution without skipping an event.

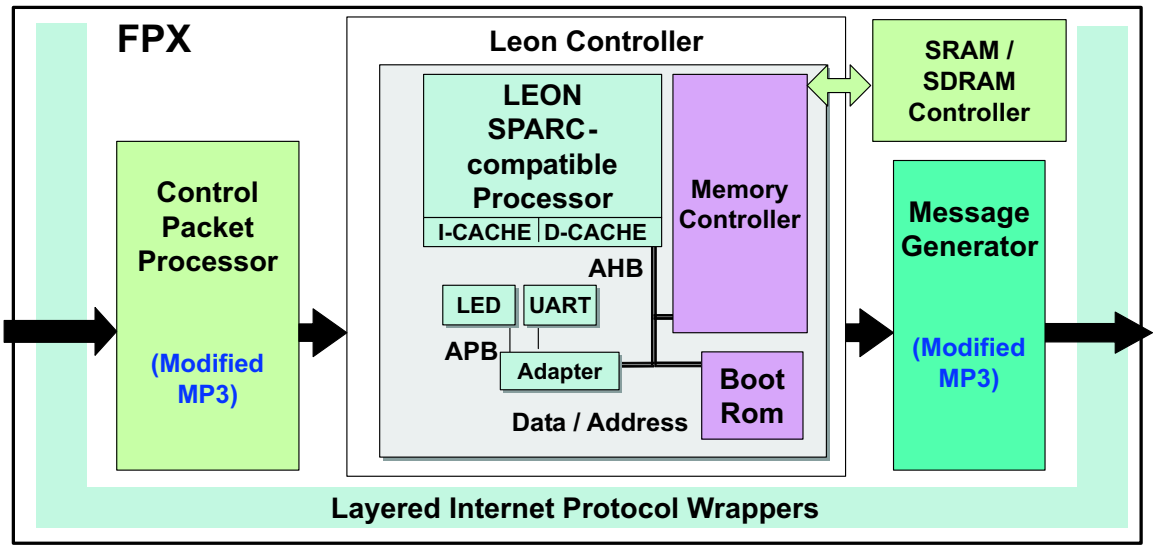


Figure 3.2: Modular diagram of the FPX and Liquid Processor Module

3.2.2 The Liquid Processor Module

The Liquid Architecture system is an extensible hardware module on a Field-programmable Port Extender (FPX) [2]. This platform is surrounded by Layered Protocol Wrappers, which parse input and formats output as **User Datagram Protocol** (UDP) network packets. Once packets are parsed, they are routed by a Control Packet Processor (CPP) which delivers certain packets with command codes to the LEON controller. The LEON controller reads these commands and directs the LEON processor accordingly, or it communicates with the memory controller to read the contents of external memory. Also present is a Message Generator which formats messages that contain command packet acknowledgements and profiling data.

3.3 Interfacing with the Hardware

The Liquid Architecture group agreed upon a web interface for access to the reconfigurable system. We designed and implemented a user-friendly interface that makes much of the system's functionality available to the web-user as well as the transparent, analyzable, profiling output.

The challenge in creating such an interface lies in toolchain support for converting a normal C-program to a Liquid Architecture-compatible binary format, communicating the user's configuration specifics to the hardware, and reporting the results back to the user for analysis.

3.3.1 Toolchain and Language Support

One of the advantages of the Liquid Architecture system is that we can execute software programmed in the C programming language. At current, our system has several restrictions on program behavior and compatibility. None of our benchmarks came into conflict with these restrictions, but nonetheless, we plan to increase the compatibility and resolve these issues in the near future:

- No file or terminal I/O. The Liquid Architecture system does not have a terminal for *printf(...)* or *getchar()* commands, so all of our programs and benchmarks reroute their trace debugging and output to SRAM memory. This memory may be read after program execution.
- No operating system calls. Our current profiling interface does not run software on top of our customized Linux kernel; this is future work for the group.
- No floating point computation. We plan to incorporate a Floating Point Unit (FPU) into the microarchitecture soon, but at current we do not offer this functionality.

Once we have a candidate .c file, we must compile, assemble, link, convert it to binary for upload onto the hardware, and create a memory map of the binary file. We use LEOX 1.0.18, an open source toolkit for cross-compiling from linux to the target LEON (SPARC V8) **Instruction Set Architecture** (ISA). We combine this toolkit with a number of other operations and use the following commands as the base of our toolchain functionality:

```
sparc-elf-gcc foo.c -s
sparc-elf-as foo.s -o foo.o
sparc-elf-ld <libraries> foo.o -o foo.out -Map foo.map
sparc-elf-objcopy foo.out -v -O binary foo.bin
```

Our customized compilation suite performs several intermediate and latter steps to produce a data file for software simulation. However, the above steps alone are sufficient to ready a C program for execution on the Liquid Architecture system via the web interface.

3.3.2 The User Interface

The web-based user interface provides the vehicle from the creation of the binary input file to the profiling results. The work of the interface can be divided into several tasks: accepting custom user programs, gathering configuration input, manipulating the hardware, and reporting the results.

Accepting C Programs

We provide two avenues to running cross-compiled programs on the Liquid Architecture system. We provide several pre-compiled regression tests and benchmarks that may be selected from a drop-down list on the first **Hypertext Mark-up Language** (HTML) configuration form (see Figure 3.3).

In the case that the user wishes to run a custom program, two file-input fields are available on the same page. The user may browse his or her local computer for the compiled binary file and its appropriate linker map file (provided by the cross-compiler toolchain, as mentioned in Subsection 3.3.1), and upload it for configuration and execution.

The screenshot in Figure 3.3 displays a text input for specifying the memory address at which to load the selected binary file. At this address, the *.text* segment will be loaded, followed by the *.data* and *.bss* segments. Further, the hardware will begin its execution at this address.

Gathering User Input

The interface permits further configuration of the execution after the binary and linker map files have been selected (see Figure 3.4). We parse the linker map file submitted by the user, then generate a web form with a grid for selecting specific memory ranges and processor events. These ranges and events will govern the initialization of the statistics module, as discussed above in Subsection 3.3.2. This permits method-wise profiling for a variety of statistics.

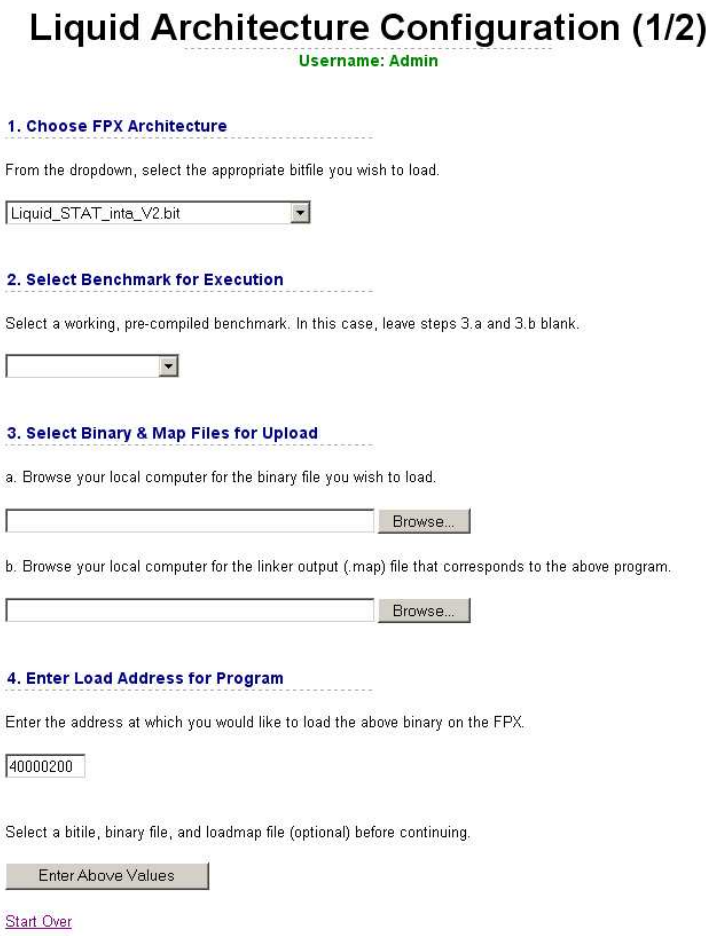


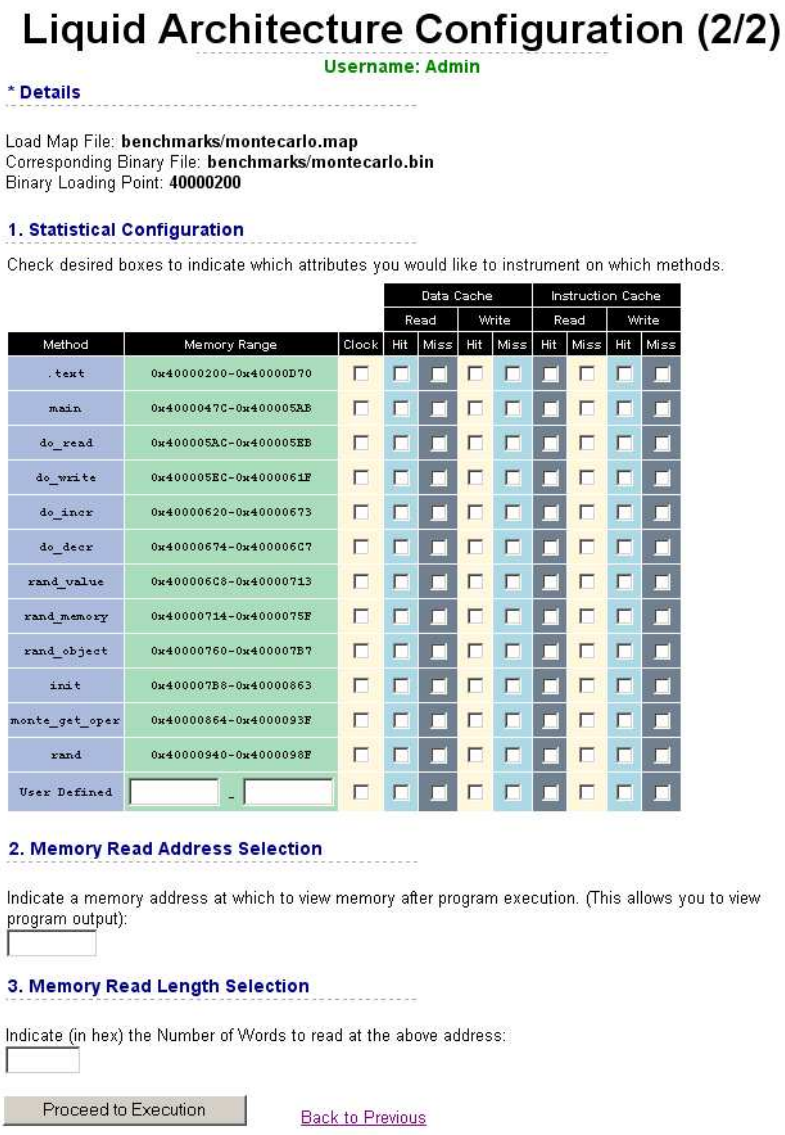
Figure 3.3: Configuration page 1 of Liquid Architecture interface

Because the Liquid Architecture system does not have a terminal or print-stream interface, our programs write to memory for output and debugging. On the second configuration page, we allow the user to specify a memory address to read after execution, making such program data available to the user.

Manipulating the Hardware

Once the configuration is complete, the user choices are posted to a Perl [13] *control center* script that communicates with the hardware using custom opcodes within UDP packets.

The server uses a Java program handle this correspondence; this program wraps the input opcodes and waits for the acknowledgements or responses. This interface



Liquid Architecture Configuration (2/2)

Username: Admin

*** Details**

Load Map File: **benchmarks/montecarlo.map**
Corresponding Binary File: **benchmarks/montecarlo.bin**
Binary Loading Point: **40000200**

1. Statistical Configuration

Check desired boxes to indicate which attributes you would like to instrument on which methods.

			Data Cache				Instruction Cache			
			Read		Write		Read		Write	
Method	Memory Range	Clock	Hit	Miss	Hit	Miss	Hit	Miss	Hit	Miss
.text	0x40000200-0x40000D70	☐	☐	☐	☐	☐	☐	☐	☐	☐
main	0x4000047C-0x400005AB	☐	☐	☐	☐	☐	☐	☐	☐	☐
do_read	0x400005AC-0x400005EB	☐	☐	☐	☐	☐	☐	☐	☐	☐
do_write	0x400005EC-0x4000061F	☐	☐	☐	☐	☐	☐	☐	☐	☐
do_incr	0x40000620-0x40000673	☐	☐	☐	☐	☐	☐	☐	☐	☐
do_decr	0x40000674-0x400006C7	☐	☐	☐	☐	☐	☐	☐	☐	☐
rand_value	0x400006C8-0x40000713	☐	☐	☐	☐	☐	☐	☐	☐	☐
rand_memory	0x40000714-0x4000075F	☐	☐	☐	☐	☐	☐	☐	☐	☐
rand_object	0x40000760-0x400007B7	☐	☐	☐	☐	☐	☐	☐	☐	☐
init	0x400007B8-0x40000863	☐	☐	☐	☐	☐	☐	☐	☐	☐
monte_get_oper	0x40000864-0x4000093F	☐	☐	☐	☐	☐	☐	☐	☐	☐
rand	0x40000940-0x4000098F	☐	☐	☐	☐	☐	☐	☐	☐	☐
User Defined	_	☐	☐	☐	☐	☐	☐	☐	☐	☐

2. Memory Read Address Selection

Indicate a memory address at which to view memory after program execution. (This allows you to view program output):

3. Memory Read Length Selection

Indicate (in hex) the Number of Words to read at the above address:

Proceed to Execution Back to Previous

Figure 3.4: Configuration page 2 of Liquid Architecture interface

is highly extensible; as our system grows in functionality and control packet sophistication, this Java program can continue to act as a communication layer between the user and the liquid processor.

The Perl control script dictates the opcodes to Java program according to the configuration specifics of the above steps. This provides a layer of abstraction between the user and the raw opcodes, requiring the user only to make the above configuration decisions and the rest executed automatically.

Each trial of the liquid architecture system shares some common functionality and opcodes for initializing and executing a given program. These steps are shown in the upper half of Figure 3.5:

- We **load the bitfile** with `ncharge`. This transmits the specifics of our reconfigurable microarchitecture over to the hardware.
- We **reset the LEON processor** by sending an opcode via our Java communication program. This wipes the memory clean and initializes the hardware to a ready state.
- After resetting, and throughout initialization, we **check processor status** by sending an opcode and receiving a status code via our Java program.
- We **upload the binary program** via our Java program, and transmit the binary data to the liquid hardware's SRAM, to a location designated by the user in Figure 3.3.
- **Configuring the statistics module** requires a series of opcode transmissions from the Perl script to the hardware. For each counter, as selected in the configuration stage shown in Figure 3.4, we have to make two transmissions to the hardware: the high and low memory addresses and the event or signal to monitor. After each counter is configured, we send an opcode to tell the statistics module how often to flush its counters and send statistics packets back, as discussed in Subsection .
- Finally, we send an opcode to **start the program**.

At this point, the program is running on the reconfigurable hardware which is configured according to the uploaded bitfile. Our control script keeps listening for processor performance feedback and program results.

Liquid Architecture Execution

Script Execution Began at: 03/30/2005 10:46:00

Username: Scott

Loading Bit File

Command Issued c WB.bit Welcome to NCHARGE, type h for help, o for option menu FPX> c WB.bit Timed OUT Command executed

Done.

Resetting

[414b3534]
eraseScript: Waiting for Lock[414b3534]
sent seqNum: 1 Leaving SendFile; sent: 460 [414b3634]
[414b3530]
[414b3534]
Run Completed

Done.

Loading Binary

Loading benchmarks/montelite.bin to 40000000
sent seqNum: 1 sent seqNum: 2 sent seqNum: 3 sent seqNum: 4 Leaving SendFile; sent: 3300 [414b3634]

Done.

Configuring Stats Module

Using .text: 40000000 - 40000C84
[414b3730]
...init'ing counter 0 to count Clock here.
java udp AC 0 0 0
[414b3730]
...init'ing counter 1 to count DCache RHit here.
java udp AC 0 1 1
[414b3730]
...init'ing counter 2 to count DCache RMiss here.
java udp AC 0 2 2
[414b3730]
...init'ing counter 3 to count DCache WHit here.
java udp AC 0 3 3
[414b3730]
...init'ing counter 4 to count DCache WMiss here.
java udp AC 0 4 4
[414b3730]
...init'ing counter 5 to count Mem Read here.
java udp AC 0 5 5
[414b3730]
...init'ing counter 6 to count Mem Write here.
java udp AC 0 6 6
[414b3730]

Done.

Configuring Feedback Timeout

java udp BC 11110000 [414b3730]

Done.

Starting Program

Starting execution at 40000000
[414b3530]

Done.

Awaiting Program Termination

Piping output to statout.txt
53500001 1110ffd5 02695c07 00017c0e 00a4311c 000052b1 0001f0af 000117a1 00000000 00000000 00000000 53500002 11110001 0268d904 00017ca8 00a40232 00004eab 0001f131 0000f873 00000000 00000000 00000000 53500003 11110001 0268e408 00017c03 00a4047e 00004ea3 0001f0f6 0000f85d 00000000 00000000 00000000 53500004 11110001 0268e07a 00017ba3 00a403e1 00004e92 0001f043 0000f764 00000000 00000000 00000000 53500005 11110001 0268ec54 00017d1a 00a4068a 00004e98 0001f18d 0000f8d2 00000000 00000000 00000000 53500006 11110001 0268eaf2 00017b25 00a4062e 00004e99 0001f012 0000f753 00000000 00000000 00000000 53500007 08b331ce 013aa73c 0000c25e 0053a689 00002807 0000fdf0 00007f67 00000000 00000000 00000000 4150504c 444f4e45

Done.

Pkt. Header	.text (Clock)	.text (DCache RHit)	.text (DCache RMiss)	.text (DCache WHit)	.text (DCache WMiss)	.text (Mem Read)	.text (Mem Write)
53500001	286,326,741	40,459,271	97,294	10,760,476	21,169	127,151	71,585
53500002	286,326,785	40,425,732	97,448	10,748,466	20,139	127,281	63,603
53500003	286,326,785	40,428,552	97,283	10,749,054	20,131	127,222	63,581
53500004	286,326,785	40,427,642	97,187	10,748,897	20,114	127,043	63,332
53500005	286,326,785	40,430,676	97,562	10,749,578	20,120	127,373	63,598
53500006	286,326,785	40,430,322	97,061	10,749,486	20,121	126,994	63,315
53500007	145,961,422	20,621,116	49,758	5,482,121	10,247	65,008	32,615
7 total	1,863,922,088	263,223,311	633,593	69,988,078	132,041	828,072	421,729

Return

Figure 3.5: Control script for Liquid Architecture execution

Gathering and Reporting the Results

As results return from the hardware, a Java listener program pipes them out to a file. Once the hardware finishes executing the program, it sends a final packet to signify either graceful termination or an error result. In either case, the server now has a file with the statistics of the program's execution on the hardware.

Our same Perl script parses this file and displays the results in a table for the user, shown in the bottom of Figure 3.5. If the user requested to read a memory address (from the screen shown on Figure 3.4), the server sends a UDP packet to the hardware requesting the values in memory at the prescribed address range. We display this data in hexadecimal, decimal, and ASCII to facilitate checking program results.

In its entirety, the automated configuration and result output of the liquid architecture takes roughly one minute. The program execution time in between configuration and output depends on the behavior of said program on our 25MHz processor.

Chapter 4

Dusty Cache

In this chapter we present our dusty data cache microarchitecture optimization and discuss its design and interaction with the machine architecture. We classify this policy as an enhancement to the write-back cache policy that is reviewed in Section 2.2. This dusty cache specification is implemented in the Liquid Architecture system (Chapter 3) as a data cache, and is analyzed in Chapter 6.

4.1 Dusty Cache Design

The dusty cache specification employs the same lines (blocks), subblocks, and valid bits as both the write-through and write-back policies. As discussed in Section 2.2, the write-back cache policy uses a dirty bit to decide when to write a value back. Our proposed dusty cache uses a *dusty check* to decide when to write the value back to memory.

The Dusty Check

The dusty check is not an actual bit as in write-back policy, but we still provide a mechanism for checking whether to write the value back to memory without checking main memory itself. Like the write-back policy, the dusty cache has a dirty bit to decide whether the value has changed since entering cache. In addition, the dusty cache has a second cache bank that acts as an image of main memory, labeled D_{Image} in Figure 4.1. This bank is readily accessible without incurring the time delay of reading main memory, discussed in Chapter 1.

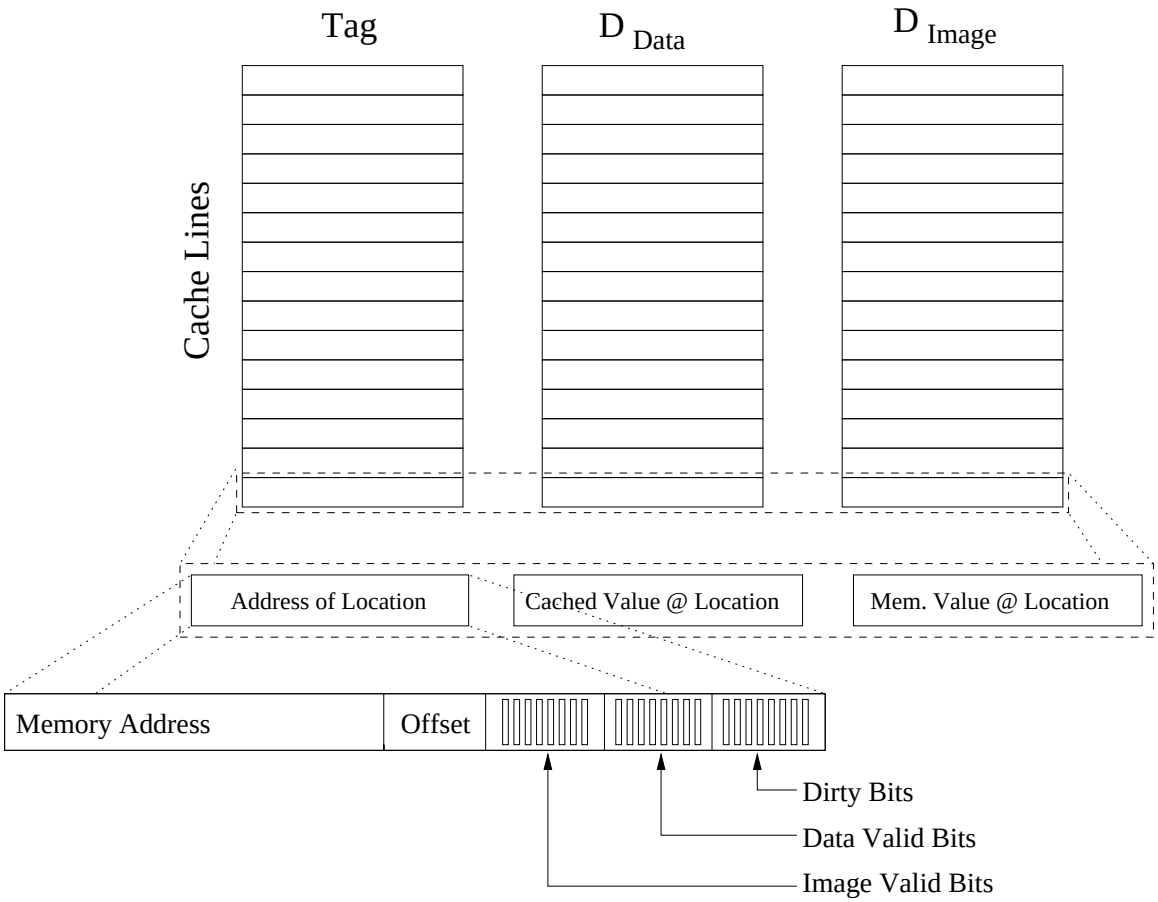


Figure 4.1: Dusty cache structural design

Dusty Cache Structures

The dusty cache policy has a single Tag RAM and a set of data lines D_{Data} like write-through and write-back, but it also has an extra set of data lines, discussed above. We maintain that for each entry in the Tag table Tag_i the corresponding line in the D_{Data} cache bank, $Data_i$, is the cached value pertaining to the address in Tag_i. The corresponding value $Image_i$ in the D_{Image} cache bank is an image of the value in memory at the address specified in Tag_i. We discuss the interaction of these corresponding elements in Section 4.2.

Both cache banks have valid bits for each subblock, but only the subblocks in D_{Data} have dirty bits. We will see why as we discuss the behavior.

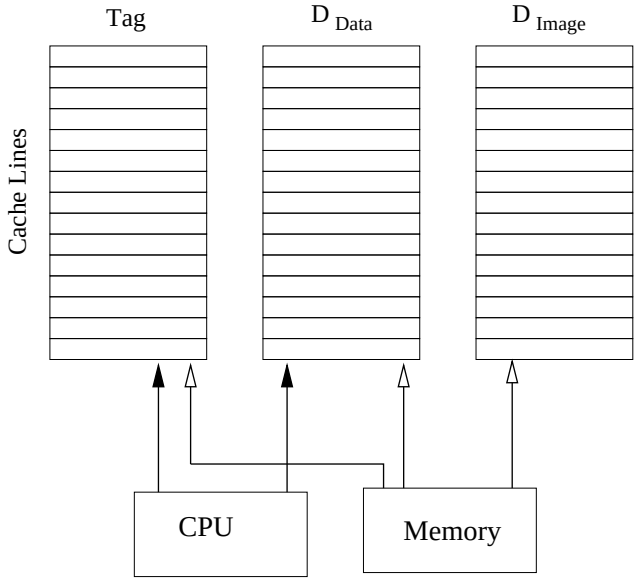


Figure 4.2: Dusty cache with reference to the CPU and main memory

4.2 Dusty Cache Behavior

Because the D_{Image} cache bank is an image of main memory, it is not written directly by the CPU in the event of a memory store; instead, only D_{Data} is written. Whenever we read from memory, however, both cache banks are written. We write to D_{Image} to retain an accurate reference of memory, and we write to D_{Data} because the CPU uses it as its data cache. This writing behavior is shown in Figure 4.2.

As we argued in Subsection 2.2.1, our proposed cache policy prevents the unnecessary memory writes incurred by write-back policy. We examine the dusty cache's behavior in several different scenarios:

- Upon a **read hit** the value is in D_{Data}, so the value is returned to the CPU.
- Upon a **read miss** the value is not in D_{Data}, so we read the value from main memory and write it to both D_{Data} and D_{Image}. Potential cache eviction.
- Upon a **write hit** the address maps to D_{Data}, so we alter the value in D_{Data} and set the dirty bit.
- Upon a **write miss** the address is not mapped to D_{Data}, so we allocate and write it to D_{Data}. Potential cache eviction.

- Upon a **cache eviction**, if the subblock's dirty bit is set, we examine the value in D_{Data} against the corresponding value in D_{Image}. If they are identical, nothing is written. Else, we write the value back to main memory.

4.3 Dusty Cache Cost

Because processor real estate is somewhat limited, we must justify our intentions of placing a memory image on a microprocessor. The addition of the memory image roughly doubles the size of our effective cache, since each subblock in our cache has a corresponding subblock in our on-chip memory image. We can imagine several objections:

- With the space that dusty cache takes on chip, it may be the case that we can better improve performance by simply doubling the size of our write-back cache, thus improving our miss rate.
- Aside from reducing the miss rate, doubling our write-back cache could potentially keep lines in our cache twice as long, and therefore reduce the rate of evictions and write-backs, both necessary and unnecessary.

Regarding the first objection, we would expect to lower our cache miss rate by doubling the size of our cache, but we see in Figure 4.3 [5] that the returns quickly diminish if we keep doubling our cache. For a cache size of 4KBytes, we could argue that doubling our cache could provide a sufficient return; with a 1MByte cache, however, we may be able to more wisely utilize our processor space.

In addition, the effectiveness of a data cache directly corresponds to the breadth of the data that the executing program tends to use. For a program that uses a small breadth of addresses, a smaller cache will suffice; if our program accesses many addresses within a wide range, a larger cache will improve performance.

Regarding the second objection, doubling the size of our write-back cache would indeed prolong a value's lifetime in cache, but for a long program run we would still eventually suffer the unnecessary write-back of evicted cache values. Regardless of its size, if we utilize a write-back cache we cannot escape these unnecessary write-backs that a program may cause. Further, just like the above, this depends directly on the set of applications we intend to run on our cache. We make a distinction of the type of programs that best utilize dusty cache opposed to write-back cache in Chapter 6.

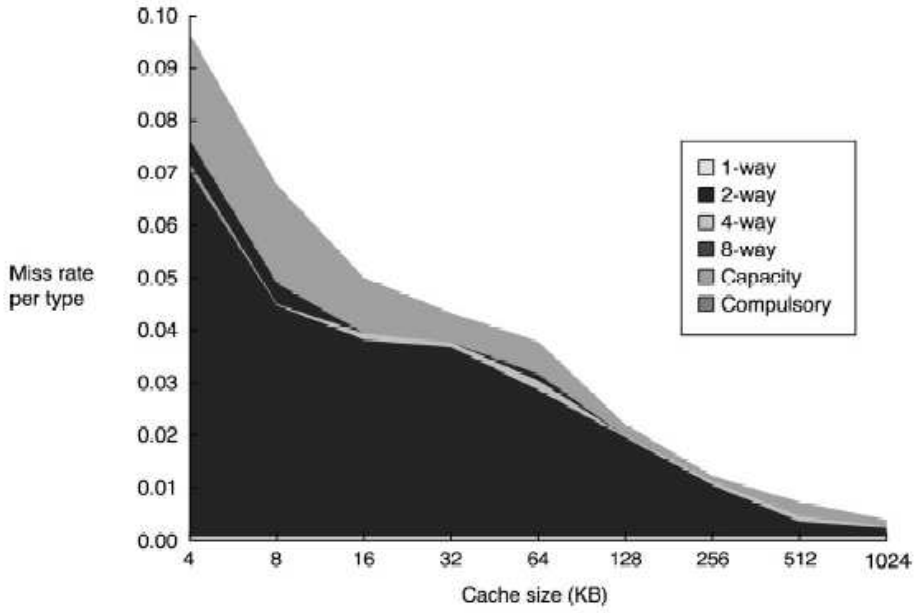


Figure 4.3: The effect of doubling our cache on cache miss rate

In further experiments, we compare these two cache policies over various benchmark executions.

In Chapter 5 we qualify the benefits of this cache design for reference counting software systems. In Chapter 6 we implement this microarchitecture and discuss the behavior of dusty cache when implemented in hardware.

Chapter 5

Software Cache Simulation

5.1 Analysis of JVM Cache Behavior

Before implementing the hardware solution to the dusty cache principle, we first profiled some common idioms of Object-Oriented Programming (OOP), embodied in some common benchmarks from the SPEC JVM '98 suite [12] to make an initial quantification of the benefits of this cache policy over write-through and write-back.

As discussed above, we expect reference counting to exhibit the efficiency of dusty cache over write-back cache, so we designed this experiment to track reference counts in Java and examine how many times the reference counts are written back to memory for different cache configurations.

JVM Instrumentation & Output

The benchmark results are gathered from an instrumented version of Sun's Java Virtual Machine 1.1.8 [9] in Solaris that implemented Reference Counting Garbage Collection [14]. To get a more transparent view of the reference counting behavior and the JVM's useage of the architecture beneath it, the JVM uses customized trace functions to signal events, such as reference count increments and decrements, `putfields`, `getfields`, and a variety of other events, complete with a dynamic count of JVM instructions at each point of execution.

The JVM outputs this data during the execution of the program, allowing us to capture runtime statistics. We ran and captured several benchmark programs, each of which is discussed in Subsection 5.1.1.

We parsed the JVM output with a trace analysis tool that constructs a graph of per-object reference count behavior. This allows us to observe the following, for every object instantiated in the benchmark:

- Its reference count at any point in execution
- The number of total JVM instructions between changes in its reference count
- The number of cache-altering instructions, such as `putfields`, `getfields`, `aastores`, and `aaloads`, that occur between changes in its reference count

It is important to note that this reference counting implementation uses a stack optimization [1]; that is, references from the stack are not tallied in an object's reference count. Instead, a single reference is made from each stack frame that contains a pointer to the object. Once this stack frame is popped, the stack reference disappears, and the object may be collected if it has has no stack frame references or object references. For this reason, we only track heap-based references in this experiment (see Chapter 6 for a stack-based reference counting traffic approximation).

Quantifying Memory Savings With JVM Output

The next task is to simulate cache memory and evaluate the cache performance for several different cache configurations. We intend to measure the efficiency of the cache in preventing writes to memory, so the metric of success in this experiment is "memory writes saved". We crafted a software solution that emulates cache behavior to gather this data.

To represent cache effectiveness, we must have a way of expressing what is currently stored in cache memory; otherwise, we cannot know which reads and writes escape the cache. We employ a probabilistic, worst-case approach here.

Whenever something is written to cache, we take the worst-case approach and assume that it will evict some value from cache. In other words, we assume that there is no locality in cache writes; every `getfield` and `putfield` instruction writes one value to cache and evicts another. From a probabilistic approach, we assume these instructions are equally likely to evict any cached value. In our implementation, this is realized as a lifetime, or "window" of cache time for each reference count value. That is, for a window of k cache writes, if value v is written to cache on write i, v will be evicted on write $i + k$.

We evaluate both unified and data (non-unified) cache configurations, and with each, we simulate write-through, write-back, and dusty policies.

- To represent the effects of unified cache (discussed in Section 2.2), we designate every JVM instruction as a cache write - this means that if reference count value v is written to cache on instruction i, it will be evicted on instruction $i + k$.
- To represent the effects of non-unified (data) cache, we monitor only events that will potentially evict a value from data cache. This includes JVM instructions getfield, putfield, aaload, and aastore, as well as reference count increments and decrements. If reference count value v is written to data cache, it will be evicted after k of these special events.

On top of the unified and data cache configurations, we simulate several cache policies: write-through, write-back, and the new dusty policy. This entails recording the number of writes to memory for each policy.

- For write-through policy, each reference count increment and decrement will be written back to memory. We use this as the frame of reference for the results of the write-back and dusty cache trials.
- For write-back policy, we adapt the above notion of the window. If a reference count enters cache memory, it is evicted after the window expires. If it has changed within the window due to an increment or decrement, it must be written back to memory (even if the value is equivalent to what it was upon entering cache). These are the only writes to memory in the write-back simulation.
- For dusty cache, we monitor savings the same way as write-back cache, save one difference: when it comes time to evict a reference count from cache, we compare its current value to its value upon entering cache. If the values are identical, it does not get written back to main memory, and this is recorded as a saving over write-back cache.

Experimental Questions

The performance of the different cache configurations will change with the behaviors of the benchmark we analyze, so we want to evaluate the performance across different types of programs. As discussed above, we expect to incur more reference count

Benchmark	Objects Created
db	8,088
javac	26,127
jess	46,129
jack	410,479

Figure 5.1: Objects created per benchmark simulated

memory traffic with programs that allocate more objects and perform fast-paced pointer changes.

In addition to investigating the performance across benchmarks, we must observe the performance across cache write policies. Specifically, within each cache write policy we want to discover which cache configuration (unified or data) is more effective.

5.1.1 Exerimental Results

We evaluated four Java benchmarks over a number of window sizes to observe memory-write savings as a function of cache sizes. In addition, we examine some statistics of each program to understand why we observed these trends.

Memory Savings per Benchmark

Each of the graphs shown above in Figure 5.2 through Figure 5.5 measure the memory writes due to reference counting that we save over write-through cache. The two dotted lines represent the percent of reference counting memory writes that write-back cache can save, and the two solid lines represent the same characteristic for our dusty cache. On all graphs, the distance between the write-back and the dusty cache measurements is the amount of unnecessary writes to memory we expect to save with dusty cache.

In Figure 5.2, we see that we can save roughly a third of all reference-counting overhead in the _209_db benchmark with a cache eviction window of size 50 if we incorporate a dusty data cache. This is roughly a 5% saving over a write-back data cache of the same size. The memory writes that the dusty policy saves over write-back policy are the "hiccups" discussed in Subsection 2.3.1.

We do not expect a great deal of savings for _209_db; this benchmark reads a 1MB data file that contains personnel records, then reads a 19KB file that contains operations to perform on the records of the data file, then performs these operations

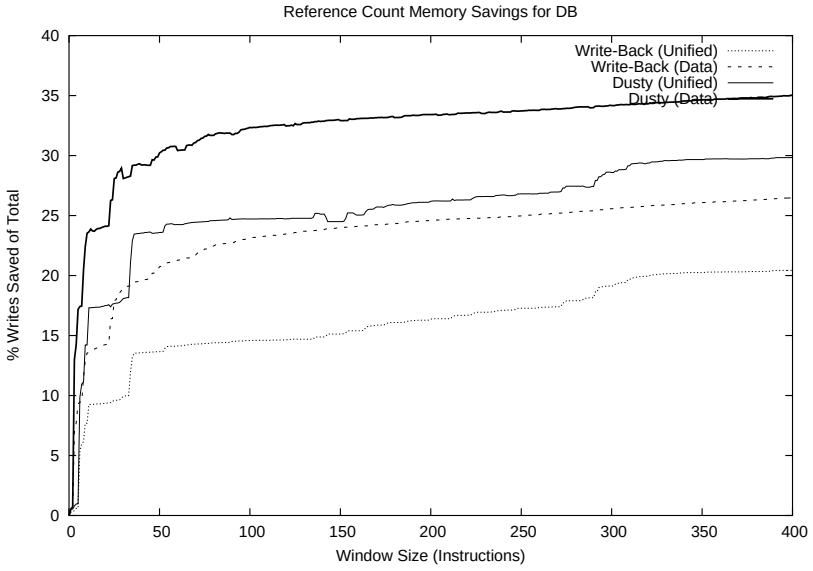


Figure 5.2: Cache simulation results for SPEC benchmark _209_DB

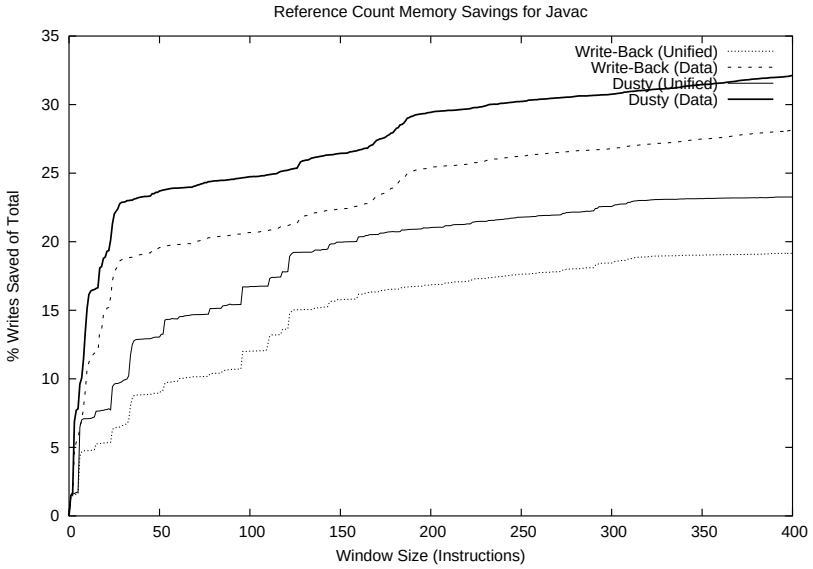


Figure 5.3: Cache simulation results for SPEC benchmark _213_Javac

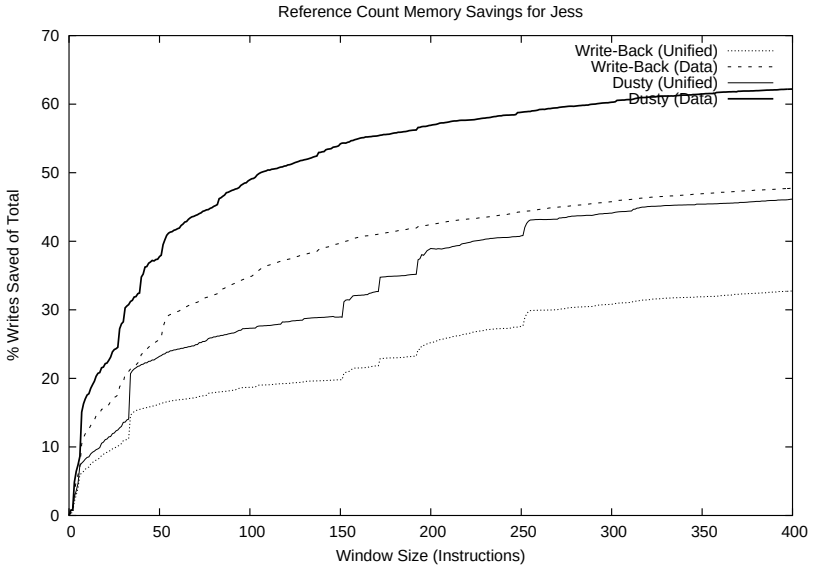


Figure 5.4: Cache simulation results for SPEC benchmark _202_Jess

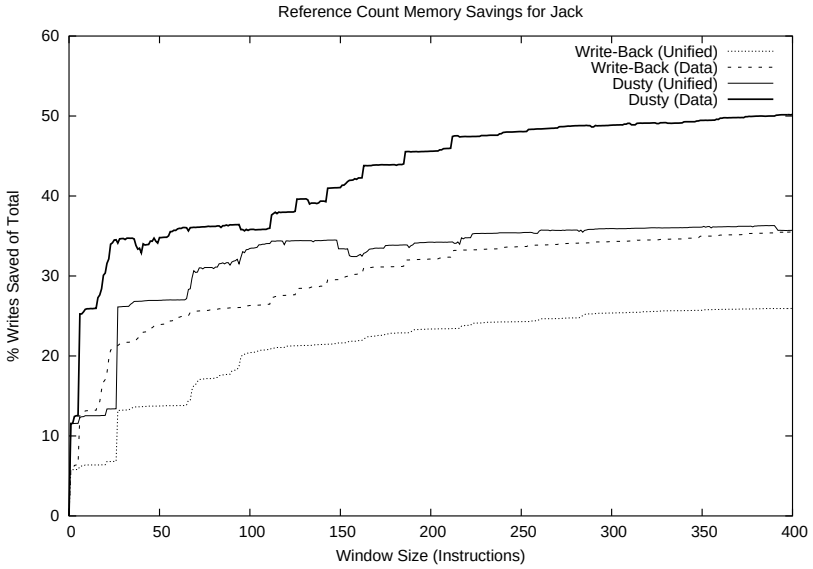


Figure 5.5: Cache simulation results for SPEC benchmark _228_Jack

[12]. In addition, we see that it only allocates 8,088 objects in total, in comparison to the other benchmarks as shown in Figure 5.1.

The Javac benchmark, shown in Figure 5.3, is the Java compiler from the JDK 1.0.2 [12]. We can see from the results that the dusty data cache implementation saves 5% of the reference counting traffic due to "hiccups." Though the benchmark itself allocates more objects than _209_db, it does not show use of quick pointer arithmetic and consequentially does not suffer heavy reference counting traffic.

The **Java Expert System Shell** (JESS) benchmark, a clone of the NASA CLIPS expert system shell shown in Figure 5.4, processes a set of *rules*, or logical "if" statements, and solves a set of puzzles [12]. The numbers for this benchmark are more interesting from a reference counting aspect. We save 25% of all reference counting traffic with a write-back cache policy in a window of 50 data cache evictions, suggesting rapid pointer manipulation. The memory-access savings from dusty data cache are more noticeable here; this may be a result of high object allocation as shown in Figure 5.1. Rougly 25% of the memory traffic savings at any window size can be attributed to preventing unnecessary write-back of reference count "hiccups."

The Jack benchmark is an early version of JavaCC, a Java parser generator with lexical analyzers. Figure 5.5 shows us a very steep slope of memory savings in the beginning, for small cache-write windows. This data and the number of objects allocated in the benchmark (as shown in Figure 5.1) suggests high-traffic object manipulation in some portion of the benchmark, and consequentially, a lot of reference counting overhead.

Results Across Benchmarks

We can easily see the difference in reference counting overhead if we examine the savings across benchmarks.

Due to the results of Figure 5.6 and the processing nature of the benchmarks, we can conclude that _213_javac and _209_db represent software that does not make use of the thumb idiom and therefore does not incur major overhead from reference counting. The benchmarks _202_jess and _228_jack suggest considerable benefit from a dusty cache implementation, and justify further investigation in hardware.

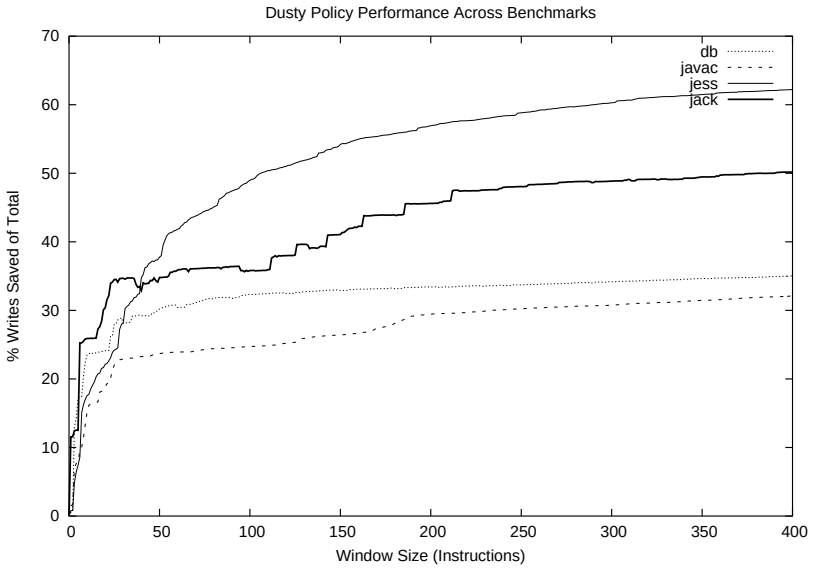


Figure 5.6: Comparison of memory-access savings due to dusty write policy across benchmarks

Chapter 6

Liquid Architecture Experimentation

After simulating dusty cache with reference counting in software, we concluded that we will save the most traffic if we implement the dusty policy in our data cache alone. After considering the simulation results of Subsection 5.1.1, we expect to contain from 30% to 50% of a program's reference counting memory traffic in cache, depending on the program's behavior.

The experiments in this chapter were conducted on the Liquid Architecture platform. The current configuration of our platform is as follows:

- We use 4 Kbytes of on-chip cache memory. For our write-back cache, this means a 4KByte data cache. For our dusty cache, this means a 4 KByte data cache and its according 4 KByte memory image.
- Our off-chip memory is SRAM and offers 4 MBytes, and we expect to have an additional 12 MBytes of SDRAM operational soon.

Because our system uses only SRAM, the ratio of cycles spent accessing cache to cycles spent accessing off-chip memory is not representative of ordinary systems. Our instrumented LEON processor can monitor read- and write-access to main memory, and report the total access of each, as discussed in Chapter 3. Counting the memory accesses will help us determine which cache policy causes more traffic to memory.

6.1 Monte Carlo Experimentation

We designed this set of trials to quantify the performance of a reference counting system with the Liquid Architecture platform. Due to the memory storage inefficiencies discussed above, we cannot yet deploy a Java Virtual Machine on the Liquid Architecture system. We therefore employ a probabilistic approach to create a benchmark that elicits microarchitecture behavior similar to that of the JVM profiled in the previous chapter.

This experiment is a *Monte Carlo* simulation [10]: it randomly triggers a set of events based on their probabilities to simulate a model. Such experiments are employed when a scenario is too difficult or expensive to evaluate analytically. Because our target benchmarks have elements that do not comply with the current Liquid Architecture platform (as discussed in Subsection 3.3.1), and because we are only interested in the cache behavior these benchmarks elicit, a Monte Carlo simulation best suits our needs.

To execute this experiment, we supply the set of events, the probabilities of each, and a framework to elicit the microarchitecture behavior of each event.

6.1.1 Determining the Set of Events

As discussed in Section 5.1, we are interested in evaluating the performance of the cache; this depends on the values resident in cache. Therefore, we are interested in monitoring only certain events that will alter the cache performance. In reference to our JVM with reference counting garbage-collection, this pertains to the following:

- *Reads*: `getfield` and `aaload` instructions.
- *Writes*: `putfield` and `aastore` instructions.
- *Heap-based RefCount++/−−*: heap-based reference points to or away from an object.
- *Stack-based Refcount++/−−*: stack-based reference points to or away from an object (approximated with `astore` instructions as discussed in Subsection 6.1.2).

From the above events, we can infer reference count increments and decrements. We next develop a mechanism to determine the relative probability of each event.

Event	Occurrences
Reads	1,182,870
Writes	209,491
AStores	197,794
Heap-Based RefCount++	67,691
Heap-Based RefCount−−	54,706

Figure 6.1: Occurrences of Cache-Altering Events

6.1.2 Determining Event Probability

We discovered in Subsection 5.1.1 that the dusty policy is a reasonable cache write policy to adopt on the data cache, most noticeably for programs with high reference counting traffic. The results in Figure 5.6 encourage us to examine the JESS benchmark to examine the probabilities of each event.

We gather the results by running the benchmark on the same instrumented JVM from our software simulation experiments. We added event-counting functionality to our trace analysis tool discussed in Section 5.1 and analyzed the JVM output. These results are shown in Figure 6.1.

It is important to note that the write and read occurrences do not include the actual read and increment of the reference count value. Rather, these counts pertain to JVM instructions that can alter the data cache.

As discussed in Section 5.1, our particular JVM reference-counting implementation [1] does not increment or decrement an object's reference count when a stack-based pointer points to or away from it. For this reason, we approximate stack-based reference-counting traffic by observing the occurrences of the `astore` instruction. We know that an `astore` will increase a reference count, but we must estimate how often the instruction overwrites a non-null value and decrements a reference count.

For a lower bound, we look at the ratio of heap-based decrements to heap-based increments (found in Figure 6.1) and multiply the value by the total number of `astores`. For our upper bound of stack-based decrements, we use the total number of `astores`. We see the results in Figure 6.2.

A similar approach to approximating reference-counting traffic on the stack is to track the `aload` instruction in addition to the `astore` instruction. For a stack-based machine, we could argue that an object's reference count would increase upon an `aload`, increase again upon the `astore`, then decrease once the operation is over. This behavior would result in even more unnecessary reference counting traffic to

Event	Occurrences
Stack-Based RefCount++	197,794
Stack-Based RefCount−−	159,851 - 197,794
Heap-Based RefCount++	67,691
Heap-Based RefCount−−	54,706

Figure 6.2: Occurrences of Reference Counting Events

Event	Probability
Read	.5727
Write	.1263
RefCount++	.1601
RefCount−−	.1409

Figure 6.3: Probability of Cache-Altering Events

memory because it exhibits the thumb idiom described in Subsection 2.3.1. Since we use a register-based architecture, we do not account for this `aload` phenomena.

After gathering the occurrences of each event, we then convert these values to relative probability format, as shown in Figure 6.3. For our immediate purposes, we will assume the stack-based reference count decrement is the median value in the range.

Now that we have the events and their relative probabilities, we construct a framework to fire these events with their respective probabilities.

6.1.3 A Framework to Model JVM Behavior

As we discussed earlier in the chapter, we cannot load our instrumented JVM onto the Liquid Architecture platform for execution due to the current platform restrictions. However, as discussed in Section 2.1, we can elicit the same microarchitecture behavior with different programs. Therefore, our objective is to develop a framework to fire actions that provide machine instructions that are similar to that of the JVM for each event discussed above. We can identify these actions for each individual event.

- A *read* is an access to a memory address whose value may or may not be cached. For this reason, we allocate an array of 1024 integers to represent the memory used by our program. Upon a memory read, we randomly access an array index and read the value at that memory address into a register.

- We execute a *write* by generating another random index into the same memory array. Instead of saving this value to a register, we simply write over it with another value.

- In the event of a reference count *increment*, we access an array of 64 integers that represent our reference count table. We generate a random index to determine which reference count to increment, read the corresponding value into a register, increment it, and write it back to its position in the array.

- A reference count *decrement* is the same as the above, except the value we write is one less than the one we read in.

The majority of the program branches off a main loop that generates a random number and selects an event based on the probabilities listed in Figure 6.3.

6.1.4 Ensuring Valid Experimental Results

The foremost challenge of this Monte Carlo approach is keeping the simulation program from contaminating the results that we wish to monitor. This can present itself as skewed results from the statistics module, or as a memory system whose contents does not reflect the data from our simulation.

As discussed above, Monte Carlo simulations require random numbers. Because we want to refrain from disturbing the simulation results, our random number generator must disrupt the memory system state as little as possible. However, our random number generation method necessarily reads and writes a single value in memory upon each new value generated. This will alter the cache, but we can take further precautions to lessen its effect on our experimental results.

The Liquid Architecture system allows us to take other measures to ensure valid results. As discussed in Subsection 3.3.2, we can perform method-wise profiling. This allows us to isolate our random number computation to a single method and refrain from explicitly tracking it in our statistics module. Therefore, though the random number computation will alter the state of the cache, the actual *cache hit* or *cache miss* event will not be tracked.

6.1.5 Monte Carlo Experimental Results

We gathered the results by following the protocol described in Section 3.3 and navigating through the configuration interface, shown in Figure 3.3 and Figure 3.4. Our

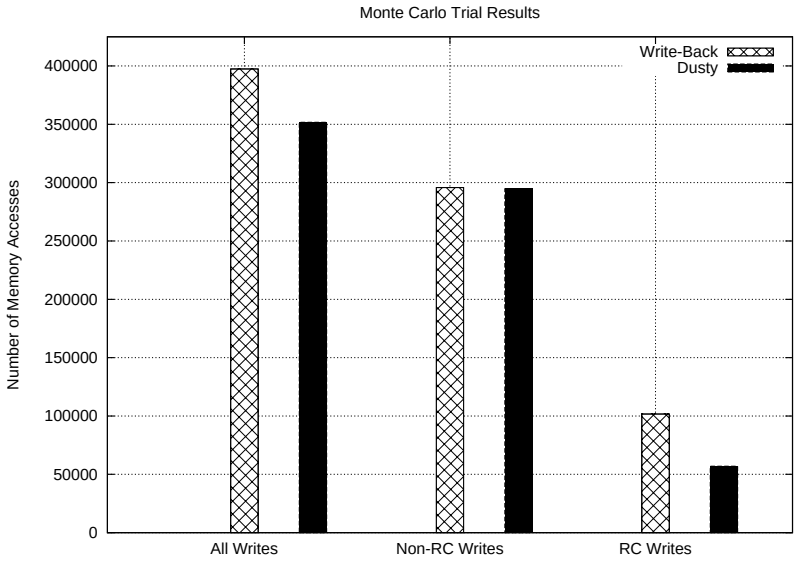


Figure 6.4: Monte Carlo benchmark results for write-back and dusty policies

primary goal is to quantify the reference-counting memory traffic that occurs in dusty and write-back caches.

For both cache policies, we observe three results: total memory writes, memory writes without reference-counting, and memory writes due to reference counting. We obtain these results by executing the benchmark twice: once with reference-counting enabled, and once with the same sequence of events but omitting the reference-counting instrumentation. We then compute the reference-counting memory writes from these two execution results as so:

$$MemoryWrites_{RefCount} = MemoryWrites_{Total} - MemoryWrites_{NoRefCount}$$

In the results shown in Figure 6.4, we find that the non-reference-counting instrumentation and read/write simulation occupies over two-thirds of the program's memory writes. Once we separate the reference-counting traffic from the rest, we find that we save roughly half of our memory writes. It follows from the data and from the design of our cache in Chapter cpt:cache that half of the memory-write traffic of write-back cache was unnecessary.

The dusty cache savings on non-reference-counting traffic is unremarkable here. The *write* event of our Monte Carlo simulation consisted of writing a random number to memory, so we find very few occurrences of value change-and-return.

In our JVM experiment analysis, we categorized dusty cache as more effective for data cache than for unified cache. In analyzing this experiment, we can conclude that when put into practice, dusty cache is more effective for some classes of data than others. We can identify a distinctive class of momentary data such as reference counting that operates better under dusty write policy than under write-back.

6.2 Dusty Cache Performance Across Benchmarks

After seeing the memory traffic savings of the dusty cache policy with a reference counting simulation, another research question arises. How much memory traffic, if any, will the dusty cache policy save for other, non-reference-counting programs?

We gathered some C benchmarks, compiled them with our cross-compilation suite, then executed them on the platform with write-back and dusty cache implementations. These benchmarks elicit a variety of memory useage patterns, as discussed below.

- *Towers of Hanoi* is a popular recursive computation puzzle. It uses no global variables or arrays; rather, it exclusively uses register computation and recursion. For this experiment, we use ten "discs" in the puzzle.
- *Numeric sort* is a heap-sort benchmark that generates an array of random integers and sorts it. We use an array of size 1,000.
- *BLASTN* (Basic Local Alignment Search Tool / Nucleotide) is a widely employed software solution for comparing genetic material. We analyze stage 1 of BLASTN in this benchmark: open-addressed, double-hashing of bases (A, C, T, G). We generate these bases randomly.

We executed these benchmarks with the same cache configurations as the Monte Carlo simulation above: 4Kbytes of cache for our write-back configuration, and 4Kbytes of cache paired with a 4Kbyte memory image for our dusty configuration. The results of the trials and the computed percent savings are shown in Figure 6.5. We can better understand these results by further examining the behavior of these programs.

Because Towers of Hanoi is almost entirely stack-based in its computational phase, it uses little RAM storage. For this reason, we see no evictions from cache

Benchmark	Write-Back Writes	Dusty Writes	Percent Saved
Numeric Heapsort	233	220	5.5%
BLASTN	185,548	180,203	3.0%
Towers of Hanoi	0	0	0.0%

Figure 6.5: Memory writes per benchmark for write-back and dusty caches

and no writes to memory. Further, this means that dusty cache cannot improve this benchmark's performance over write-back.

As mentioned above, the numeric heapsort benchmark generates an array and sorts it. This involves swapping values from one array index to another, where some of the array values may be equivalent. Because this array is not stack-based, we see more memory writes than Towers of Hanoi. The five percent savings with dusty cache suggests that heapsort occasionally swaps a value away and then back to a location prior to an eviction.

BLASTN uses an open-addressed, double-hashed hashtable to organize nucleotides, so change-and-return behavior is possible in this stage. We can imagine an enhanced statistics module that tracks the method that tracks the instruction memory address that last wrote a cached value. This data would help us identify the method in which a saving occurred upon an eviction from dusty cache. At current, this module is in development. Without it, we are still able to quantify the memory access savings for these benchmarks and answer the question at the beginning of Section 6.2.

6.3 Extrapolating Dusty Cache Performance

We measure and report the results of our Liquid Architecture experiments in both *memory writes* and *percent memory writes saved*. This metric is more portable than *nanoseconds saved* or *clock cycles saved* because the ratio of the Liquid Architecture system's processor and memory speeds are far different than that of a modern desktop computer. Moreover, clock cycles measure cache speed and average performance rather than the behavior of the cache policy.

By reporting our results in memory writes, we can project clock cycle savings for different memory and processor speeds. In Figure 6.6 we demonstrate the effect of increasing the processor speed from its current speed of 25MHz up to 2.5GHz. We

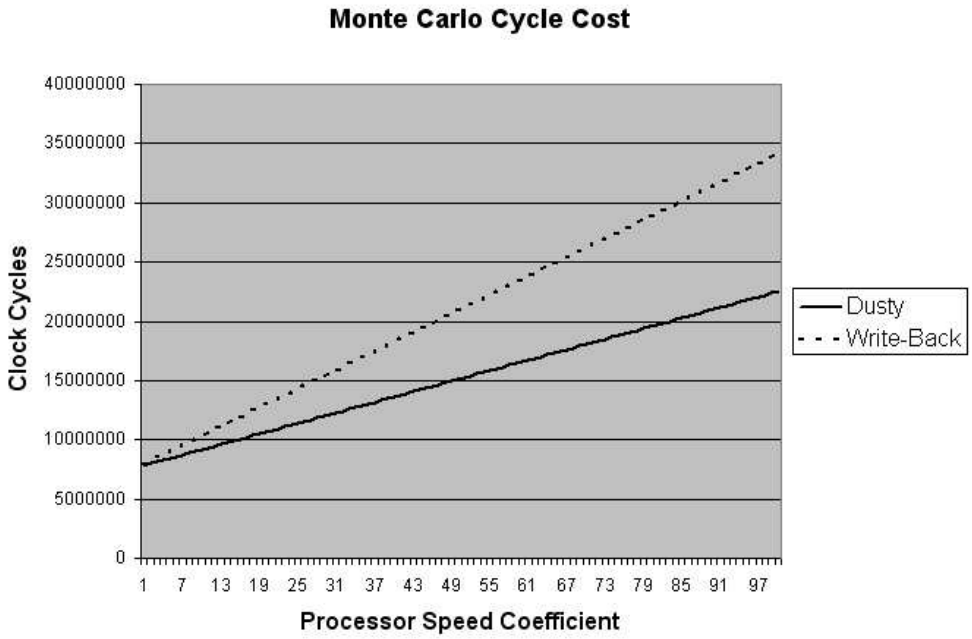


Figure 6.6: Monte Carlo (reference counting portion) cost in processor clock cycles as processor speed increases

estimate this by changing the clock cycle cost of a memory write such that if our processor were 50MHz, twice as many clock cycles would pass during a write stall.

We computed Figure 6.6 using the "Reference Counting Only" data set from our Monte Carlo simulation, shown on the far right of Figure 6.4. This permits us to see how many clock cycles the processor dedicated to incrementing, decrementing, reading, and writing reference counts. From the figure, we see that if our processor were one hundred times faster, we would expect to save roughly one-third of the clock cycles spent on total reference counting overhead with dusty cache.

We expect to save even more clock cycles with faster systems, due to the "memory wall" notion discussed in Chapter 1.

Chapter 7

Conclusion

In this chapter we review the progression of this thesis and the experimental results. From this, we illustrate some contributions and future work that may follow from these results.

7.1 Thesis in Review

We designed and implemented the dusty cache write policy, and we present experimental results that show its effectiveness in executing the garbage-collection technique of reference counting.

We first approximated the memory savings with an instrumented JVM and a cache approximation program. From these results, we discovered that the dusty cache policy saves the more memory writes as a data cache than as a unified cache, and is more effective for programs of greater object traffic. We further discovered that we can save 50% to 70% of the total reference counting traffic to memory with our dusty cache.

We implemented the dusty cache policy in VHDL and realized it with the Liquid Architecture platform. We then created a Monte Carlo simulation in C to run on this platform. We tailored this experiment to elicit similar microarchitecture behavior to a JVM that uses reference counting garbage-collection.

Due to the reconfigurability of the Liquid processor, we were able to attain cycle-accurate performance measurements of the dusty cache performance as well as write-back cache performance. We measured the writes to memory with both caches in a Monte Carlo simulation. Our results show that dusty cache saves unnecessary

memory traffic due to reference counting in addition to other value-change-and-return behavior in normal program flow.

Through our experiments, we have qualified the efficiency of dusty cache in practice: it is an effective data cache for momentary change-and-return data values. Though we primarily explore reference counting as our example of this type of data, we make the case that this cache write policy effectively prevents said writes to memory in any instance where a cached value changes and returns to its former value prior to eviction.

7.2 Future Work

This thesis has quantified the memory traffic savings of dusty cache and illustrated its effectiveness for different types of data and cache configurations. This opens several avenues of further dusty cache research.

- Dusty cache may work best as an exclusive data cache, explicity for momentary change-and-return data such as reference counts. In this respect, the compiler may route normal data traffic through a general write-back cache, but defer reference counting instrumentation to a block of memory buffered solely by a dusty cache. The research would entail finding an optimal compromise of caches to conserve chip space as well as prevent unncessary writes to memory.

- Similar to the above, we could segregate the dusty cache and optimize a compiler to track all possible change-and-return values and route them to the dusty-buffered memory space.

- In general reference counting implementations, we see only increments and decrements; we never see a reference count overwritten by an identical value. The dusty cache will prevent the ensuing unnecessary write to memory, yet this *identical overwrite* gesture was not measured in this thesis. This gesture may be common in other scenarios and merits investigation.

References

[1] Morgan Deters, Nicholas A. Leidenfrost, Matthew P. Hampton, James C. Brodman, and Ron K. Cytron. Automated Reference-Counted Object Recycling for Real-Time Java. In *Proceedings of 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2004.

[2] Field Programmable Port Extender Homepage. Online `http://www.arl.wustl.edu/arl/projects/fpx/`, August 2001.

[3] Jiri Gaissler. The LEON processor. `www.gaisler.com`, 2005.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[5] John L. Hennessey and David A. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, 2003.

[6] Richard Hough and Ron K. Cytron. The liquid architecture statistics module. Tech Report, 2005.

[7] SPARC International. *The SPARC Architecture Manual Version 8*. Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[8] Phillip Jones, Shobana Padmanabhan, Daniel Rymarz, John Maschmeyer, David V. Schuehler, John W. Lockwood, and Ron K. Cytron. Liquid architecture. In *Workshop on Next Generation Software (at IPDPS)*, 2004.

[9] Tom Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1997.

[10] N. Metropolis. The beginning of the Monte Carlo method. Los Alamos Science, 1987.

[11] David V. Schuehler, Benjamin C. Brodie, Roger D. Chamberlain, Ron K. Cytron, Scott J. Friedman, Jason Fritts, Phillip Jones, Praveen Krishnamurthy, John W. Lockwood, Shobana Padmanabhan, and Huakai Zhang. Microarchitecture optimization for embedded systems, 2004.

[12] SPEC. Specjvm98 benchmarks. `www.spec.org/osg/jvm98`, 1998.

[13] Larry Wall. *Programming Perl, 3rd Edition*. O'Reilly, Sebostopol, CA, 2000.

[14] Paul R. Wilson. Uniprocessor garbage collection techniques. International Workshop on Memory Management, 1992.

[15] W.A. Wulf and S.A. McKee. Hitting the memory wall: Implications of the obvious. In *ACM Computer Architecture News*, pages 20–24. 1995.

Vita

Scott J. Friedman

Date of Birth May 9, 1980

Place of Birth St. Louis, Missouri

Degrees Master of Science, Computer Science
Washington University in Saint Louis, May 2005

Bachelor of Science Cum Laude, Computer Science
Washington University in Saint Louis, May 2003

Publications Scott J. Friedman, Nicholas A. Leidenfrost, Benjamin Brodie, Ron K. Cytron. **Hash Tables Ready for Real-Time**. In *IEEE Realtime and Embedded Systems 2001 Workshop*, December 2001

May, 2005