

Dusty Caches for Reference Counting Garbage Collection *

Scott Friedman
sfj1@cse.wustl.edu

Praveen Krishnamurthy
praveenk@cse.wustl.edu

Roger Chamberlain
roger@cse.wustl.edu

Ron K. Cytron
cytron@cse.wustl.edu

Jason E. Fritts
jefritts@cse.wustl.edu

Department of Computer Science & Engineering
Washington University
St. Louis, MO 63130

ABSTRACT

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 cannot detect all garbage on its own, it is well suited for some applications and is implemented typically in conjunction with other methods to increase 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 investigate a cache design that takes advantage of temporarily silent stores, by remembering 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.

1. INTRODUCTION

In this paper, we examine the effectiveness of a write-back cache policy that can eliminate some writes to memory. One scenario in which this policy can be effective is reference counting, a garbage-collection technique that can perform well [11], except for the introduction of excessive memory traffic to maintain reference counts. Reference counting maintains a count of pointers that reference every object. In many cases, a reference count changes from its value v , but then quickly returns to v . Normal write-back cache policy marks the count *dirty* upon the first value change, ensuring its 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.

In this paper, we examine the extent to which an enhanced write-back cache policy can reduce the cost of reference counting. Our

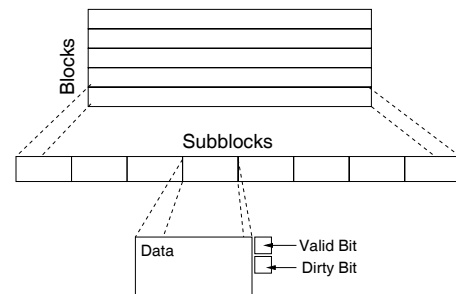


Figure 1: Write-back cache organization

experiments use the Java SPEC Benchmarks and the Liquid Architecture platform we have developed to implement and quantitatively analyze the microarchitecture optimization at clock-cycle resolution.

In terms of related work, we essentially investigate a form of *silent stores* [1, 9]—in particular, *temporally* silent stores [10]—based on memory traffic induced by reference-counting [2]. Our contribution lies in demonstrating the extent to which reference-counting traffic is temporally silent, in providing a design based on write-back caches, and in deploying that design on reconfigurable hardware.

While others [9, 10] consider more efficient and effective implementations for store squashing, we consider a simpler but more costly approach of duplicating L1 cache and squashing stores to RAM via a modified write-back policy. However, reference-counting traffic can be segregated from other storage traffic at compile-time, and reference-counting values can be implemented as very small nonnegative integers. Thus, we can observe temporally silent stores in an area where its benefits may be concentrated and more highly effective. Moreover, many avoid reference counting because of the extra storage traffic it induces; our contribution shows how much that traffic can be reduced by observing temporally silent stores. Our design is based on the write-back cache, whose organization is shown in Figure 1.

Consider the following situation in which a write-back is unnecessary even though the relevant subblock is dirty. If a value is altered and promptly returns to the same value in a subsequent write, it is still marked dirty and is written back to main memory even though the value in main memory is identical. Each store is not silent [1] because the stored value is different, but the cumulative

*This work was sponsored by NSF under grant ITR-0313203

effect of the stores is temporally silent [10]. The dirty bit is thus sufficient but not necessary to indicate whether a value needs to be written back to memory. We investigate a potentially more effective cache design that verifies and squashes some temporally silent stores.

2. REFERENCE COUNTING AND OOP

Reference counting [2] is an efficient albeit inexact means of automatic memory management, otherwise known as garbage-collection [16]. 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 can perform well in real-time systems [3] and has been shown to be efficient for Java [11].

Though the implementation is straightforward, reference counting can impose considerable overhead due to increased memory traffic. Examples of said traffic are found in common **Object-Oriented Programming** (OOP) patterns that have one object point to another for a short time, before pointing away. The Iterator pattern [6] is a very simple and frequently deployed example of this behavior.

2.1 The Thumb Idiom

More generally, we define the *thumb idiom* as the following sequence of operations:

1. a pointer references an object
2. Based on that pointer, a relatively quick computation is performed
3. the pointer moves on to point to another object.

For example, 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.

A common use of the Iterator pattern is shown below, where we traverse an entire list and process each item in the collection.

```
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} . The thumb touches every node of the collection, generating two reference-counting transactions per node. We next examine the effect of such activity on a data cache.

2.2 Effect on the Cache

With a write-through cache policy, both the increment and the decrement will be written directly to memory as 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

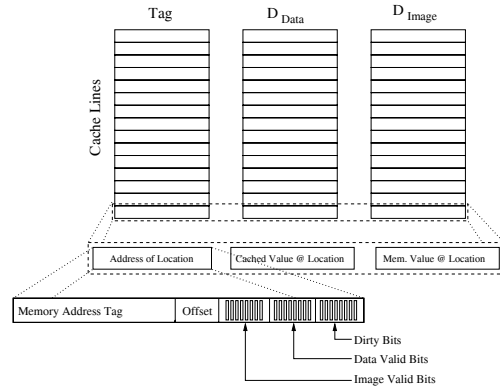


Figure 2: Dusty cache structural design

the reference count is the same before and after the short *hiccup*, the write-back cache has marked it dirty. Therefore, upon eviction from cache, the value is written to memory, even though the cached value is the same as what is stored in 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 experience no writes to memory for such hiccups—just a single read from memory to load 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 discuss the implementation of this idea in Section 3.

3. DUSTY CACHE

In this section 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 a standard write-back policy. This dusty cache specification is implemented in the Liquid Architecture system (Section 4) as a data cache and is analyzed in Section 6.

3.1 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. 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 (in the sense of a “dirty bit”), but is instead a mechanism for deciding if the cached value duplicates what was fetched from storage initially. Like the write-back policy, the dusty cache has a dirty bit to decide whether the value has changed since entering the cache. In addition, the dusty cache has a second cache bank that acts as an image of main memory, labeled D_{Image} in Figure 2. This bank is readily accessible without incurring the time delay of reading main memory, discussed in Section 1. In our implementation, we actually duplicate the data cache to realize the image; in systems offering L2 cache, that layer could serve as the image if it can be accessed sufficiently quickly.

This dusty check occurs upon cache eviction, discussed below in Subsection 3.2.

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 3.2.

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.

3.2 Dusty Cache Behavior

Because the D_{Image} cache bank is an image of main memory, it is never written directly by the CPU in the event of a memory store; instead, only D_{Data} is written. Whenever the CPU reads from memory, however, both cache banks are written. We update D_{Image} to retain an accurate reference of memory, and we write to D_{Data} because the CPU uses it as its data cache.

Our proposed cache policy is designed to prevent 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} . This can result in a cache eviction.
- Upon a **write hit** the value is in D_{Data} , so we alter the value in cache and set the dirty bit. We do not alter the value in D_{Image} .
- Upon a **write miss** the value is not in D_{Data} , so we write it to D_{Data} . This can result in a cache eviction.
- Upon a **cache eviction**, if the subblock's dirty bit is set, we compare the value in D_{Data} against the corresponding value in D_{Image} . If they are identical, nothing is written. Otherwise, we write the value back to main memory if the valid bit is set.

4. THE LIQUID ARCHITECTURE SYSTEM

The Liquid Architecture [8] 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 have been prototyped, tested, and analyzed at the circuit level without undue time or cost.

In this section, we describe the features of the Liquid Architecture project that were used to conduct experiments for this work.

4.1 The Liquid Processor Module

The Liquid Architecture processor began as LEON [5], a standard Sparc ISA for embedded systems, developed by the ESA (European Space Agency). The LEON core provides typical microarchitecture 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 Liquid Architecture system is an extensible hardware module on a Field-programmable Port Extender (FPX) [4]. 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



Figure 3: Photograph of the FPX

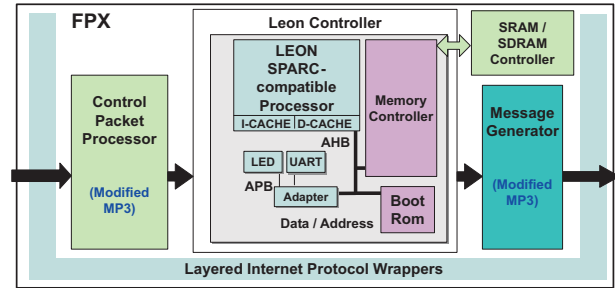


Figure 4: Modular diagram of the FPX and Liquid Processor Module

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 opcode acknowledgements and profiling data.

4.2 The Statistics Module

We modified the core to add the *Statistics Module* [14], 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-consuming to obtain through simulation. By comparison, the Liquid Architecture processor runs programs at full (FPGA) speed.

This module is implemented as a collection of smaller counter modules, each of which offers 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) that represent a program-counter range in which the event should be counted
- 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 program-counter 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 statistic 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.

5. INITIAL EVALUATION

Before implementing the hardware solution to the dusty cache principle, we first examined some popular benchmarks (the SPEC JVM '98 suite [15]) to make an initial quantification of the reference-counting benefits of this cache policy over write-through and write-back.

As discussed above, we expect reference counting to affect cache in such a way that dusty cache will save on storage traffic. We therefore 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.

5.1 JVM Instrumentation and Output

The benchmark results are gathered from an instrumented version of Sun's Java Virtual Machine 1.1.8 [12] in Solaris that implemented a variation of Reference Counting Garbage Collection [2, 16]. We instrumented the JVM to provide customized traces for events of interest, 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 as well as customized programs that implemented common programming patterns, such as linked list iteration example presented in Section 2.1.

We parsed the JVM output with a trace analysis tool (of our own devising) 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 particular reference counting implementation uses a stack optimization [3] that obviates the need to manipulate reference counts based on (JVM) stack and register activity. References from the stack are not tallied in an object's reference count. Instead, a single reference is made from the last-to-be-popped 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 no (other thread) stack frame references or heap references. The heap references are counted

using the traditional reference counting. For this reason, we only track heap-based references in this experiment (see Section 6 for a stack-based reference counting traffic approximation).

While our results would look better if we accounted for the stack activity, much of that traffic would not be present if the program were executed in native code on a typical RISC register machine. For example, to reference a field of an object, the JVM bytecode architecture requires pushing the object pointer from a register on stack. After the field is referenced, the stack cell is popped. On a RISC machine, the register could be used directly with no need to develop another, temporary pointer to the object. Thus, the reference-counting approximation seems a more appropriate environment in which to evaluate our idea.

5.2 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 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

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

Figure 5: Objects created per benchmark simulated

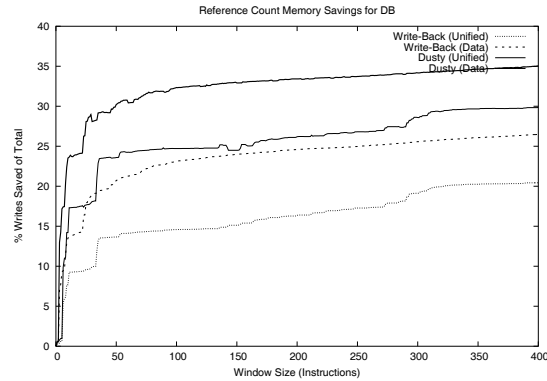


Figure 6: Cache simulation results for SPEC benchmark _209_DB

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.

5.3 Experimental 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. In Figure 6, 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 2.1.

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 [15]. In addition, we see that it only allocates 8,088 objects in total, in comparison to the other benchmarks as shown in Figure 5.

The Javac benchmark, shown in Figure 7, is the Java compiler from the JDK 1.0.2 [15]. 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

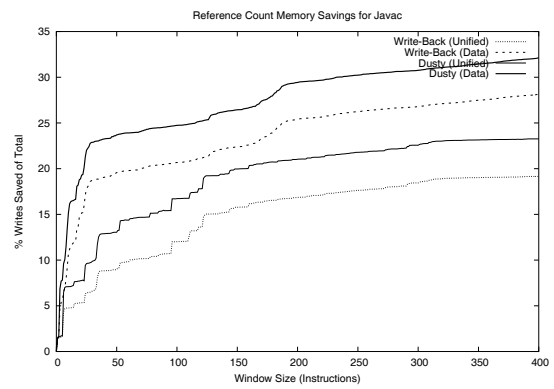


Figure 7: Cache simulation results for SPEC benchmark _213_Javac

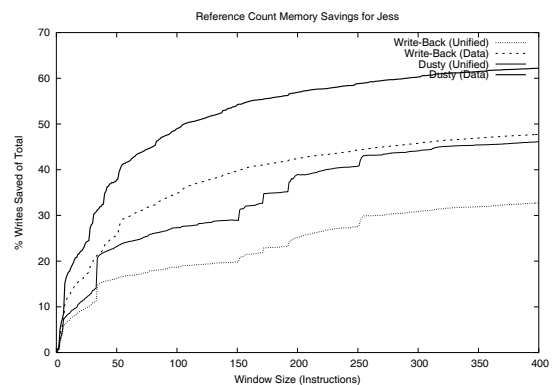


Figure 8: Cache simulation results for SPEC benchmark _202_Jess

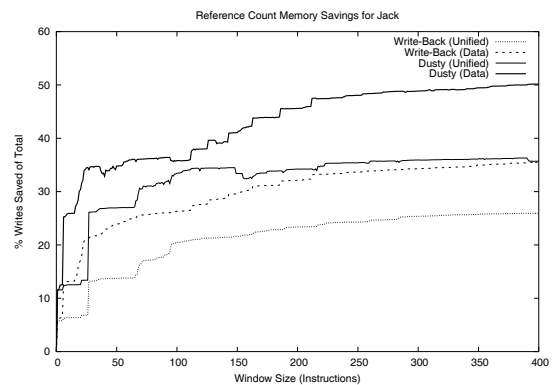


Figure 9: Cache simulation results for SPEC benchmark _228_Jack

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 8, processes a set of *rules*, or logical “if” statements, and solves a set of puzzles [15]. The numbers for this benchmark are more interesting

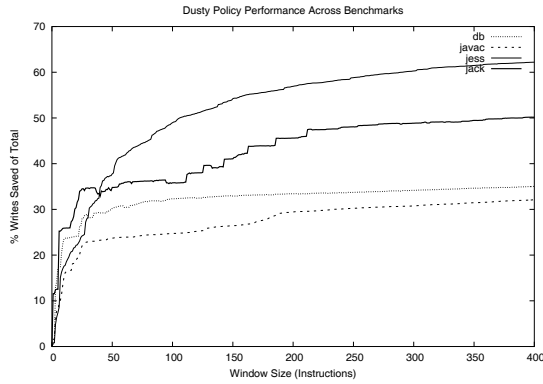


Figure 10: Comparison of memory-access savings due to dusty write policy across benchmarks

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. Roughly 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 9 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) 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 10 and the processing nature of the benchmarks, we can conclude that `_213_javac` and `_209_db` represent software that does not have heavy pointer arithmetic 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.

6. EXPERIMENTATION ON LEON

After considering the simulation results of Section 5.3, we expect to contain from 30% to 50% of a program’s reference counting memory traffic in cache, depending on the program’s behavior.

To see the results of our design in action, we implemented the dusty cache in the context of the Liquid Architecture platform described in Section 4. We point out the following characteristics of the platform in its current form:

- Off-chip memory is currently SRAM and offers only 4 MBytes.
- SRAM is relatively fast, taking only two cycles to complete a write operation. We therefore present results in terms of the number of accesses saved. As the distance between memory and the CPU increases, the performance obtained by not writing to storage also increases.

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

Figure 11: Occurrences of Cache-Altering Events

We designed a set of trials to quantify the performance of a reference counting system with the Liquid Architecture platform. Because we could not as yet deploy a Java Virtual Machine on with our Liquid Architecture, we employ a probabilistic approach to create a benchmark that elicits microarchitecture behavior similar to that of the JVM profiled in the previous section.

This experiment is a *Monte Carlo* simulation [13]: 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 and because we are only interested in the cache behavior these benchmarks elicit, a Monte Carlo simulation suits our needs.

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

6.1 Determining the Set of Events

We are interested in evaluating the performance of the cache; this depends on what values are 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 Section 6.2).

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.

6.2 Determining Event Probability

We discovered in Section 5.3 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 10 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 11.

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.

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 12: Occurrences of Reference Counting Events

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

Figure 13: Probability of Cache-Altering Events

As discussed in Section 5, our particular JVM reference-counting implementation [3] 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 11) 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 12.

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 memory because it exhibits the thumb idiom described in Section 2.1. 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 13. 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 at their respective probabilities.

6.3 A Framework to Model JVM Behavior

As we discussed earlier in this section, we cannot load our instrumented JVM onto the Liquid Architecture platform for execution due to the current platform restrictions. However, we can elicit similar 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 128 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 13.

Aside from deciding the probability and the implementation of our increment and decrement operations, we have to create a framework for deciding *which* “object” to increment or decrement. As said above, we use an array of integers to represent object reference counts; however, we have found through previous implementations of this benchmark that the strategy of deciding *which* integer to increment or decrement can profoundly alter the results.

In a normal reference-counted program, we expect all reference counts to start at one upon an object’s creation, vary somewhat during program execution, and return to one or zero by the end of the program or object collection. We adopted the following policy to determine which objects to decrement. When we increment an object, we immediately schedule it for a decrement after a window of time t_{window} in the future such that $0 < t_{window} < Window_{max}$. Upon each iteration of the simulation, we decrement any scheduled reference counts.

6.4 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. While our platform measures exactly the number of cache misses seen by the executing program, the program itself is executing and has some effect on the data cache. To mitigate the program’s effects, we wrote the program so as to cause as much storage as possible to be allocated on the runtime stack and in registers—both are implemented as structures separate from storage in the LEON core.

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 4.1, 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.

6.5 Monte Carlo Experimental Results

We ran the above program with the realized microarchitecture discussed in Section 3 on the Liquid Architecture platform. We leverage the Liquid Architecture Statistics Module to gather and return relevant data.

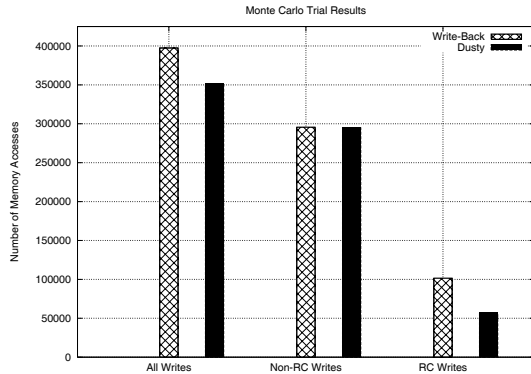


Figure 14: Monte Carlo benchmark results for write-back and dusty policies

For each cache policy, 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. In our implementation, this involved commenting out two lines, but leaving the surrounding instrumentation. We compute the reference-counting memory writes from these two execution results by subtracting: $Writes_{RefCount} = Writes_{Total} - Writes_{NoRefCount}$.

The results are shown in Figure 14 were gathered with a write-back cache size of 4KB and a dusty cache of size 4KB (4KB data, 4KB image). 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 Section 3 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.

7. CONCLUSION AND FUTURE WORK

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 use reference counting as our vehicle for analyzing cache effectiveness in this paper, and examine common programming idioms in which write-back cache policy and reference counting causes unnecessary writes to memory. Though we primarily explore reference counting in this paper, 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.

We utilized the Liquid Architecture platform to realize our proposed microarchitecture. This platform allowed us rapidly to develop and test our cache policy as well as quantify its memory traffic and performance at clock cycle level with normal C benchmarks.

The results of instrumented JVM experiments suggest that the dusty cache is more effective as a data cache than as a unified or instruction cache. However, it may be the case that it would be even more effective on a subset of the data cache, such as an exclusive cache for reference counts. This would follow in the intent of this paper: biasing the microarchitecture to take advantage of common programming idioms.

Acknowledgements

We thank Rob LeGrand for his help in crafting the LEON-deployed random-number generator used in this paper.

8. REFERENCES

- [1] Gordon B. Bell, Kevin M. Lepak, and Mikko H. Lipasti. Characterization of silent stores. In *PACT '00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, page 133, Washington, DC, USA, 2000. IEEE Computer Society.
- [2] George E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, 1960.
- [3] 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.
- [4] Field Programmable Port Extender Homepage. Online <http://www.arl.wustl.edu/arl/projects/fpx/>, August 2001.
- [5] Jiri Gaissler. The leon processor. www.gaisler.com, 2005.
- [6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
- [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] Kevin M. Lepak and Mikko H. Lipasti. Silent stores for free. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 22–31, New York, NY, USA, 2000. ACM Press.
- [10] Kevin M. Lepak and Mikko H. Lipasti. Temporally silent stores. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 30–41, New York, NY, USA, 2002. ACM Press.
- [11] Yossi Levanoni and Erez Petrank. An On-the-fly Reference Counting Garbage Collector for Java. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 367–380. ACM Press, 2001.
- [12] Tom Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, 1997.
- [13] N. Metropolis. *The beginning of the monte-carlo method*. Los Alamos Science, 1987.
- [14] Ron K. Cytron Richard Hough. The liquid architecture statistics module. Tech Report, 2005.
- [15] SPEC. *Specjvm98 benchmarks*. www.spec.org/osg/jvm98, 1998.
- [16] Paul R. Wilson. Uniprocessor garbage collection techniques. International Workshop on Memory Management, 1992.