Evaluating Dusty Caches on General Workloads

Praveen Krishnamurthy
Washington University
St. Louis, Missouri
praveen@wustl.edu

Roger D. Chamberlain
Washington University
St. Louis, Missouri
roger@wustl.edu

Ron K. Cytron
Washington University
St. Louis, Missouri
cytron@wustl.edu

Jason Fritts
Saint Louis University
St. Louis, Missouri
jfritts@slu.edu

ABSTRACT

Silent stores, i.e., stores to memory that write the same value as already stored in that memory location, have been observed to occur frequently. These stores not only create redundant memory transactions, but in a multiprocessor environment result in redundant communication messages.

In this paper, we quantify the frequency of such silent stores, in particular temporally silent stores, in typical benchmarks such as MiBench and CommBench, and evaluate the effectiveness of a "dusty" cache policy on these workloads. The dusty cache policy remembers the initial values loaded into cache from memory, and squashes temporally silent stores. We present results from experiments that compare this dusty cache policy to a standard write-back cache of comparable size.

One aspect of this work is that the evaluation environment is not simply an isolated application executing on a target architecture, but rather includes a complete run-time environment, including the OS. The effectiveness of dusty caches is assessed in the context of a full multitasking system. The empirical measurements are made using dedicated hardware on a soft-core implementation of a SPARC V8 ISA deployed on an FPGA.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*; C.4 [**Performance of Systems**]: Measurement Techniques

General Terms

Design, Experimentation, Measurement, Performance

This research is supported in part by NSF grant CNS-0313203.
WDDD '06 Boston, Massachusetts USA
.

Keywords

Temporally silent stores, performance evaluation

1. INTRODUCTION

The relative cost of a main memory access is worsening; that is, the speed of on-chip compute operations and cache memory is increasing faster than the speed of the main 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 state that "clearly, there is a processor-memory performance gap that computer architects must try to close" [7].

The above suggests that we perform our operations on-chip as often as possible, opposed to climbing the "memory wall" [21] and suffering the lower speeds of buses and memory. The most common remedy to this problem involves dedicating (more) on-chip area to the cache and optimizing its behavior to absorb more of the main memory traffic.

In this paper, we examine the effectiveness of a write-back cache policy that can eliminate some writes to memory. In [5], we introduced a new method of determining how "soiled" a value is, saying that a value is *dusty* when it is altered in cache, but dirty only when it differs from the value in memory. In terms of related work, we essentially investigate a form of *silent stores* [2, 11, 12]—in particular, *temporally* silent stores [13]. 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 [2] because the stored value is different, but the cumulative effect of the stores is temporally silent [13]. The dirty bit is thus sufficient but not necessary to indicate whether a value needs to be written back to main memory. We investigate a potentially more effective cache design that verifies and squashes some temporally silent stores. Our design is based on a classic write-back cache whose initial organization is

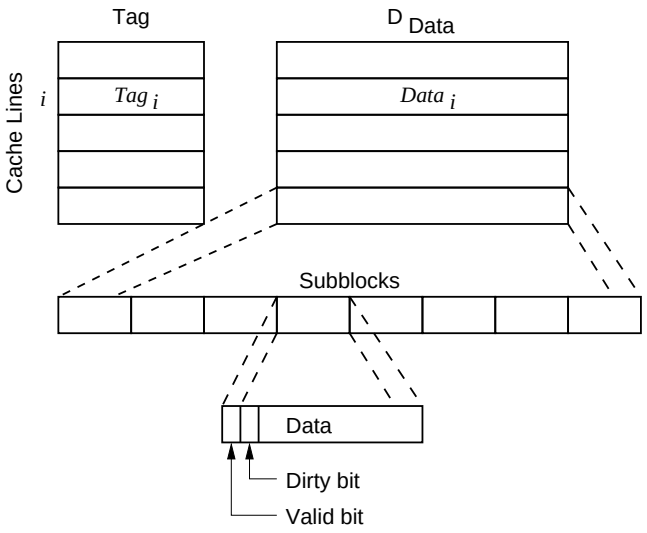

Figure 1: Write-back cache organization.

shown in Figure 1.

Our primary contribution consists of a performance analysis of dusty cache mechanisms across a range of benchmarks selected to complement the results available in the literature. In the previous work on temporally silent stores [13], two application sets were investigated. The first, from SPLASH-2 [20], represents scientific workloads, and the second consists of commercial workloads from [3]. Both of these application sets are typical of desktop and/or server systems. In addition, the focus of the earlier work was on the performance implications for multiprocessor systems.

The benchmark set we use is focused on embedded applications executing on uniprocessors. We use 3 applications from MiBench [6] and 4 applications from CommBench [19]. The MiBench suite is specifically designed to be representative of a wide range of embedded applications, and the CommBench suite is designed to be representative of networking applications (e.g., within the routers themselves).

Our second contribution is in highlighting the importance of including sufficient execution context when evaluating the performance of applications. We explicitly include a full multitasking execution environment, including the operating system. This is in contrast to many simulation-based performance evaluations in which the application of interest is executed in isolation on the architecture under investigation. Here, we compare measurements made in isolation with measurements made in the context of an operating system and demonstrate the differences that exist between the two. As is also the case in [13], the execution environment we use for performance evaluation is explicitly designed to accurately model typical usage patterns.

2. 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 3) as a data cache and is analyzed in Sections 4 and 5.

2.1 Dusty Cache Design

The dusty cache employs the same lines (blocks), subblocks, and valid bits as both traditional write-through and write-back policies. The write-back cache policy uses a dirty bit to decide when to write a value back to main memory. Our proposed dusty cache uses a *dusty check* to decide when to write the value back to main 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, and does not impact the the access time of D_{Data}. 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. Lepak [11] describes a number of alternative implementation strategies.

This dusty check occurs upon cache eviction, discussed below in Section 2.2.

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 2.2.

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.

2.2 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 a 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}.

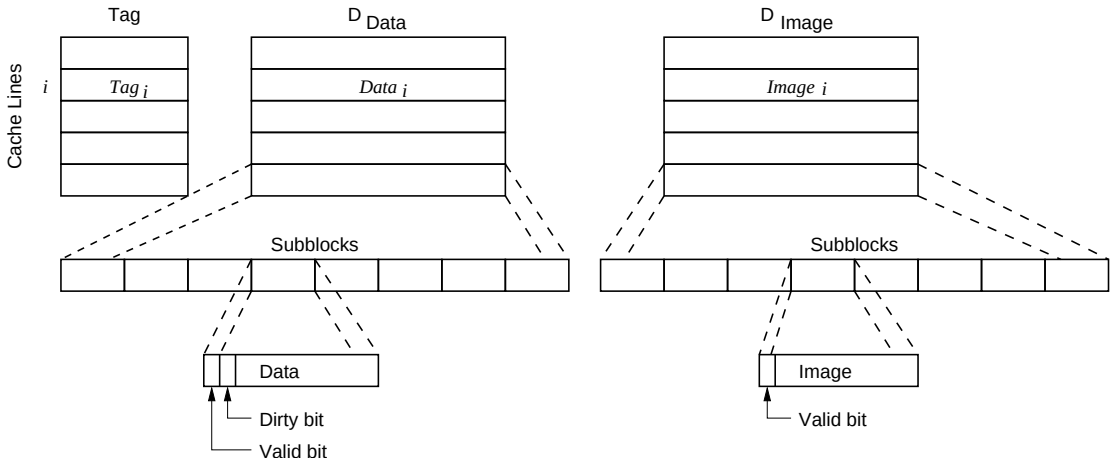

Figure 2: Dusty cache structural design.

- 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.

3. THE LIQUID ARCHITECTURE SYSTEM

The Liquid Architecture [9] 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.

3.1 Profiling

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. Feedback on such software-microarchitecture interaction 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 most cases, the profiling software adds extra overhead and provides a faulty report of processor activity. Other times, software profiling does not provide sufficient resolution of performance information so the results are too vague to draw conclusions. Simulations can provide better detail, but they can take an extremely long time to evaluate the simplest of programs. Moreover, many software profilers and simulators do not account for (or cannot adequately model) some of the rare or less probable events that occur during normal execution such as memory stalls, dynamic scheduling, operating system interactions, multithreading effects, or external interrupts.

The Liquid Architecture solution combines reconfigurable logic with a soft core processor, adding microarchitecture support for monitoring on-chip events and a web-based configuration and analysis interface. This system offers an effective solution to the above profiling problems enabling real-time, cycle-accurate performance analysis and permitting rapid design and testing of hardware and software structures. This infrastructure was used to provide results for this paper.

3.2 The Liquid Processor Module

The Liquid Architecture processor began as LEON [10], a standard SPARC V8 ISA for embedded systems, developed by the European Space Agency. Illustrated in Figures 3 and 4, the LEON processor provides typical microarchitecture features such as instruction and data caches, the entire SPARC V8 instruction set [17], and buses for high-speed memory access (the AHB) and low-speed peripheral control (the APB) [1].

We have deployed the LEON processor on the Field-programmable Port Extender (FPX) platform [14]. The FPX provides an environment where FPGA designs can be interfaced with external memory and a high-speed network interface. OS support includes both uClinux [18] when the memory management unit (MMU) is absent and Linux kernel 2.6.x when the MMU is present [4, 16]. In this work, we limit the investigation to the case without an MMU and use uClinux as the OS. A similar investigation which includes the MMU is ongoing.

3.3 The Statistics Module

We modified the LEON core to add the *Statistics Module* [8], a microarchitecture-level performance-measurement system 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 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:

- Connections to the address bus, event bus, and an output data bus

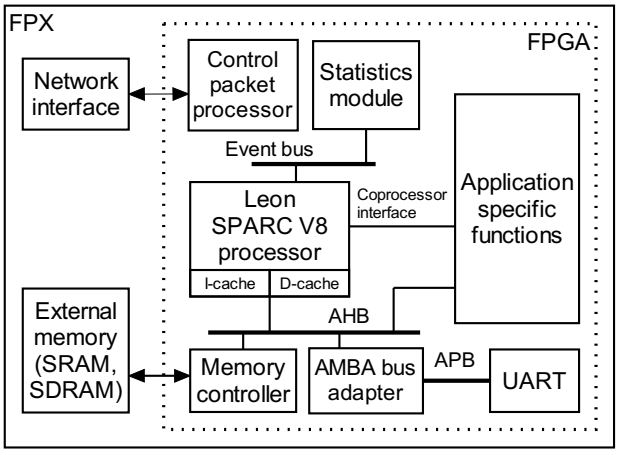

Figure 3: Liquid architecture block diagram.

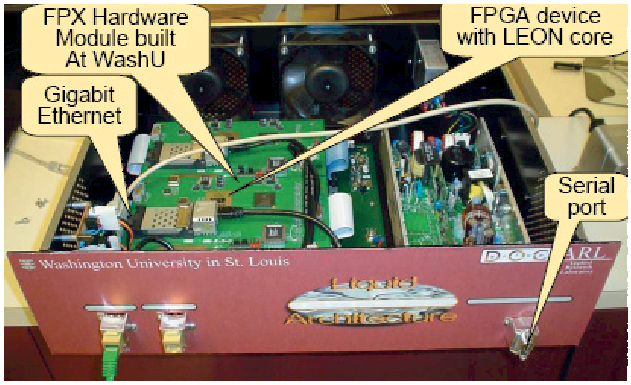

Figure 4: Liquid architecture photograph.

- 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

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 statistics module with a simple change to the 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.

An important precaution the Statistics Module takes is overflow prevention. When a user-designated amount of clock cycles expire, the 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.

4. EXPERIMENTAL METHODS

In this section we describe the benchmark set and the experimental procedure.

4.1 Benchmarks

The MiBench benchmark suite [6] consists of a set of embedded system workloads which differ from standard desktop workloads. The applications contained in the MiBench suite were selected to capture the diversity of workloads in embedded systems. For the purposes of this study, we chose workloads from the networking and automotive sections of the suite.

CommBench [19] was designed with the goal of evaluating and designing telecommunications network processors. The benchmark consists of 8 programs, 4 of which focus on packet header processing, and the other 4 are geared towards data stream processing.

Following are the set of applications we have used as part of this study:

- From MiBench:
  - *basicmath*: This application is part of the automotive applications inside MiBench. It computes cubic functions, integer square roots and angle conversions.
  - *dijkstra*, *sha*: These are part of the networking applications inside MiBench. Dijkstra computes the shortest path between nodes in a graph, and sha is a secure hash algorithm which computes a 160-bit digest of inputs.
- From CommBench:
  - *drr*, *frag*: These are part of the header processing apps inside CommBench. The drr algorithm is used for bandwidth scheduling for large numbers of flows. Frag refers to the fragmentation algorithm used in networking to split IP packets.
  - *reed_enc*, *reed_dec*: These are part of the packet processing applications in CommBench. They are the encoder and decoder used in the Reed-Solomon forward error correction scheme.

4.2 Procedure

For each of the applications, the following sequence of actions was taken:

- The application was executed standalone (i.e., no OS) on the Liquid Architecture system. For benchmarks that require disk-based input data, the benchmark was altered to either read compile-time initialized data or synthetically generate the input data. This step is required because there is no file system available for standalone execution. In addition, the benchmark was adapted to initiate the statistics collection subsystem (this required the addition of a single call at the beginning of the code).
- The modified application was also executed under the uClinux OS. Even though we have a file system for this case, our desire to study the impact of applications running with and without an OS motivated us to use the identical (altered) applications from the standalone runs.

- A number of different configurations of the LEON processor were generated. Cache sizes of 1, 2, 4, and 8 Kbytes were included for a traditional direct-mapped, write-back cache, and cache sizes of 1, 2, and 4 Kbytes were included for a dusty cache. For the dusty cache, the sizes above do not include the D_{Image} memory, so the actual on-chip memory usage for a dusty cache configuration is twice that listed above. That is, the listed cache size is the amount visible to the processor, D_{Data}.

- Each of the applications was executed on each of the processor configurations, measuring loads, stores, cache hits, cache misses, memory reads, and memory writes. It is the decrease in memory writes for a dusty cache that we wish to examine.

5. PERFORMANCE RESULTS

Table 1 shows the total number of loads and stores for each of the benchmark applications.

Table 1: Total number of load and store instructions for each benchmark.

Benchmark	Loads	Stores
basicmath	74,151,136	46,577,732
bitcnts	68,009,977	15,626,007
dijkstra	60,985,966	8,925,084
drr	180,171,104	92,676,773
frag	178,058,884	96,099,734
reed_enc	149,063,969	62,261,586
reed_dec	208,593,061	89,553,034
sha	424,476,497	158,084,970

5.1 Standalone Execution

The initial performance results are presented for the applications running standalone. Figure 5 shows, for each application and each cache size, the total count of memory writes that occur in an individual execution. Given that this is a write-back cache, these writes to the memory subsystem occur primarily as a result of cache evictions. This represents the baseline memory write traffic with the write-back cache, which is what we hope to improve with the dusty cache.

Figure 6 shows the savings in memory writes (as a percentage of the original number of memory writes shown in Figure 5) for each application and cache size when using a dusty cache. Note that the on-chip memory requirements have doubled for the dusty cache implementation, so this comparison only makes sense when in a design environment where this extra memory requirement isn't critical. This might be the case, for example, in a multi-level cache system, where the D_{Image} memory is actually implemented as part of the next level in the memory hierarchy. Additional implementation techniques that do not explicitly require double the memory are described in [11]. An alternative way to view this figure is that it is a measure of the frequency of silent stores that potentially can be squashed, irrespective of the cost of the method.

What is noteworthy here is the number of cases where the savings are quite substantial, frequently well above 80%. This is an even greater frequency of silent stores than that

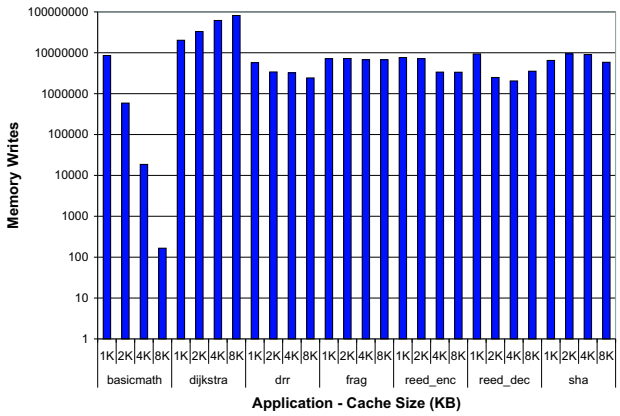

Figure 5: Count of memory writes for traditional write-back, direct-mapped cache for each application and cache size. The applications are executing standalone.

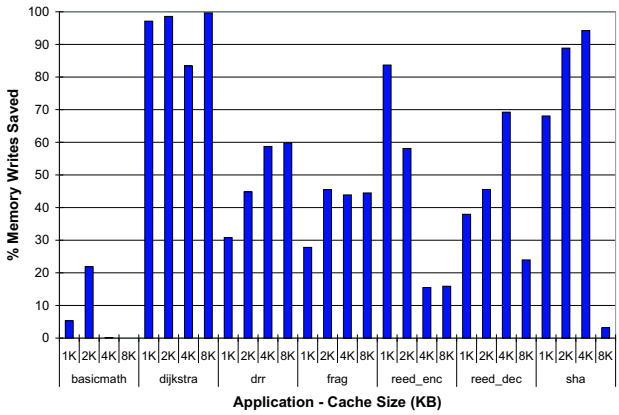

Figure 6: Percentage of memory writes saved with a dusty cache. The applications are executing standalone.

reported in [13]. The surprisingly large fraction of silent stores in the dijkstra benchmark is due to the repeated traversals of the graph, most of the time writing values that are already present.

When constructing a dusty cache system where the D_{Image} memory doesn't come for free, a more appropriate comparison is between a write-back cache of some size and the dusty cache of half that size. This fairly adjusts the comparison for the actual memory usage of the dusty cache implementation (accounting for both D_{Data} and D_{Image} memory requirements). This is shown in Figure 7.

Here, we clearly see different tradeoffs playing out in distinct applications. For example, the drop in memory writes saved for the basicmath benchmark is reasonable given the initial number of memory writes (as shown in Figure 5) is so strongly cache size dependent.

Whether or not using half of the available on-chip memory space to build a dusty cache is now application dependent, with potentially large swings in performance either direction. In addition, the overall performance of the application

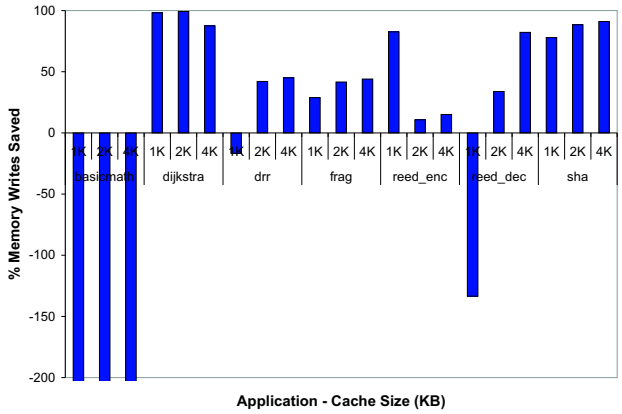

Figure 7: Percentage of memory writes saved with a dusty cache of comparable memory requirements. The applications are executing standalone. Note that the y-axis has been truncated at -200%, and the actual values for those exceeding -200% are well below.

will now be impacted in other ways by the difference in cache size. For example, Figure 8 plots the miss rate for reads as the cache size varies. Clearly, all of these factors will have to be considered before one can reliably choose the best performing configuration. Under these circumstances, whether or not to use a dusty cache is yet another microarchitectural tuning knob for systems such as [15].

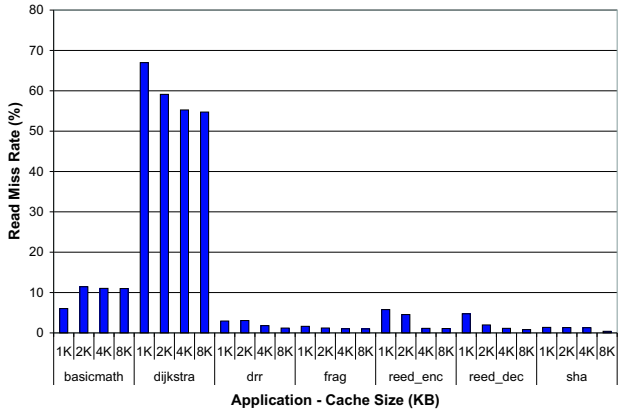

Figure 8: Data cache miss rate for processor load instructions. The applications are executing standalone.

Returning to the assumption that the cost of the D_{Image} is not prohibitive, we next consider an alternative control mechanism. Figure 9 shows the writeback savings at the full block level, rather than the subblock level of Figure 6. Under the assumption that memory transactions occur for complete cache lines, the dusty cache will save a memory write only when the entire cache line is either not dirty or matches the contents of D_{Image}. As can be seen in the plot, the savings are qualitatively very similar to the case where decisions are being made at the individual subblock level. We attribute the improvements seen in some cases (e.g., drr, reed_enc, and reed_dec for both 4 KB and 8 KB cache sizes) to write locality.

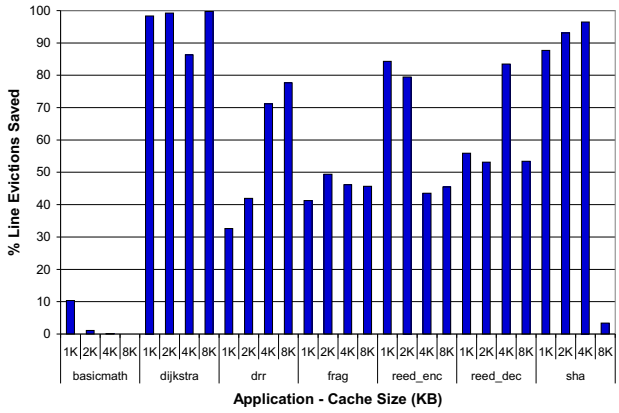

Figure 9: Percentage of line evictions saved with a dusty cache. The applications are executing standalone.

5.2 Execution with the Operating System

When running the benchmark applications on the OS, we do not separately measure memory operations for the application and the OS, but rather measure aggregate memory writes from all sources. Figure 10 plots the number of memory write operations for each application and cache size when running on the uClinux OS. Note that in virtually all cases, the memory performance is noticeably different (including both increases and decreases in memory writes) than the standalone case.

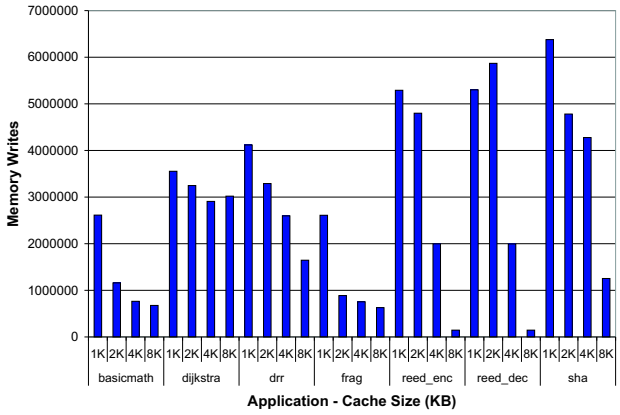

Figure 10: Count of memory writes for traditional write-back, direct-mapped cache for each application and cache size. The applications are executing on the OS.

Following in the pattern of the previous subsection, we next show the percentage of memory writes saved with a dusty cache implementation that is of the same size D_{Data} as the write-back cache. This is illustrated in Figure 11.

While the particular results are distinct from the standalone execution, the savings are still surprisingly large.

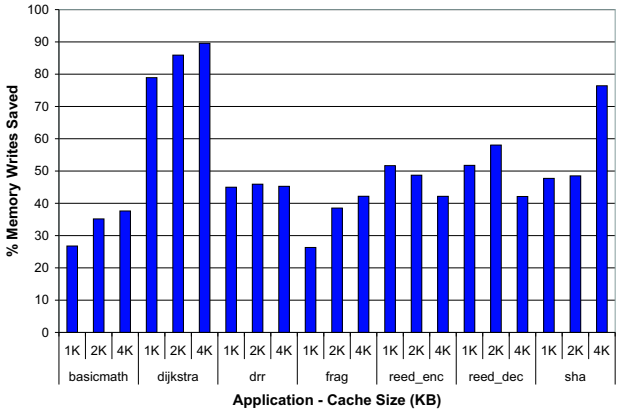

Figure 11: Percentage of memory writes saved with a dusty cache. The applications are executing on the OS.

We next plot the savings in memory writes when the total memory usage for the write-back cache is equivalent to that of the dusty cache. This is shown in Figure 12. Once again, the tradeoff is application specific, but the answer as to whether a larger traditional cache or a smaller dusty cache has better memory performance is dependent on whether or not the application is running on the OS.

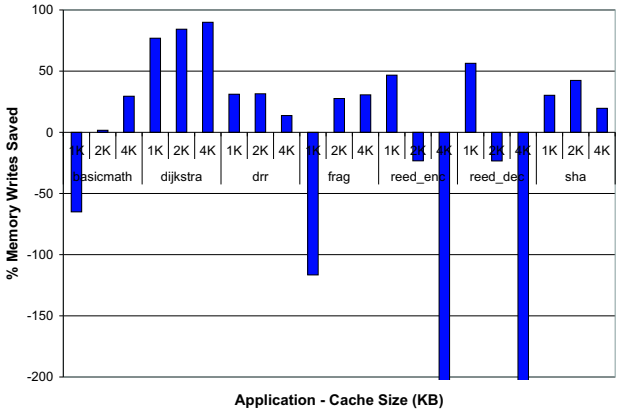

Figure 12: Percentage of memory writes saved with a dusty cache of comparable memory requirements. The applications are executing on the OS.

Maintaining the symmetry with Section 5.1, we next present (in Figure 13) the data cache miss rates for reads when the benchmarks are being executed on the OS. Again, we see similar qualitative patterns, but clear quantitative distinctions when running standalone or with an OS. We hypothesize that for cases that have improved miss rates with the OS (e.g., basicmath, dijkstra) the improved miss rates within the OS proper are amortizing the poor miss rates in the application itself.

Finally, Figure 14 shows the savings due to the dusty cache policy if writeback decisions are being made at the

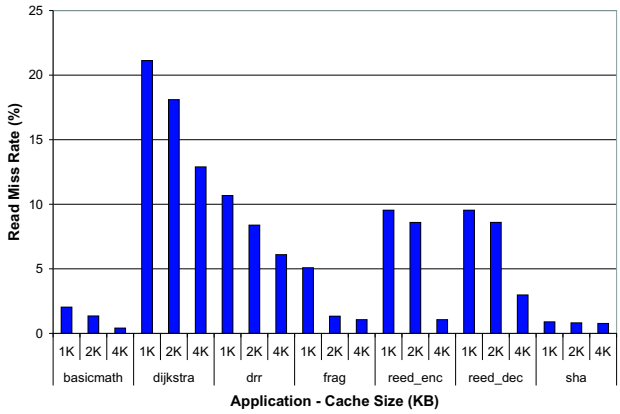

Figure 13: Data cache miss rate for processor load instructions. The applications are executing on the OS.

full block level rather than the subblock level. An interesting observation to be made here is that there is generally better similarity between word evictions and line evictions (i.e., comparing Figure 6 to Figure 9 and Figure 11 to Figure 14) than there is similarity between executing standalone and with the OS (i.e., comparing Figure 6 to Figure 11 and Figure 9 to Figure 14).

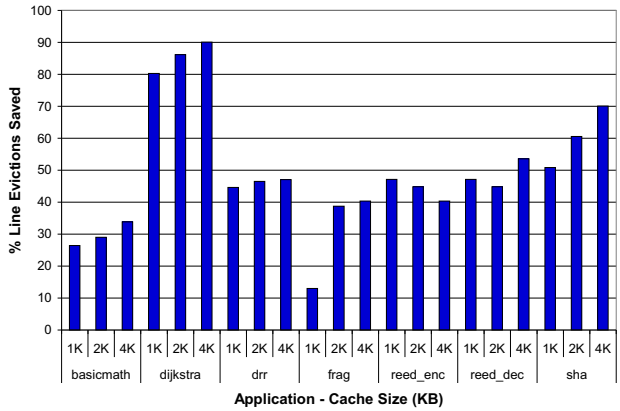

Figure 14: Percentage of line evictions saved with a dusty cache. The applications are executing on the OS.

6. CONCLUSIONS

There are a number of conclusions that can be drawn from the above data. First, temporally silent stores are frequently occurring. The embedded benchmarks examined here exhibit even greater frequency of silent stores than previously published results (which focused on scientific and commercial workloads). Second, when deciding between a dusty cache system and a traditional write-back cache of comparable memory usage, the properties of the particular application(s) to be executed are critical. Finally, the above decision cannot be made by examining the application in isolation, but rather must include the impact of the run

time system, as that can change the end result as to which approach will have better performance.

The significant differences seen in the results with and without the OS have motivated us to further our investigations in this area. The Liquid Architecture measurement infrastructure enables us to probe more deeply into the properties of the execution, and we plan to use this capability to better understand when standalone execution measurements can and cannot be trusted for performance evaluation purposes.

7. REFERENCES

[1] AMBA Specification (Rev 2.0). ARM, Ltd., `http://www.arm.com/products/solutions/AMBA_spec.html`.

[2] G. B. Bell, K. M. Lepak, and M. H. Lipasti. Characterization of silent stores. In *Proc. of Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 133–142, Oct. 2000.

[3] H. W. Cain, R. Rajwar, M. Marden, and M. H. Lipasti. An architectural characterization of Java TPC-W. In *Proc. of 7th Int'l Symp. on High Performance Computer Architecture*, Jan. 2001.

[4] R. D. Chamberlain, R. K. Cytron, J. E. Fritts, and J. W. Lockwood. Vision for liquid architecture. In *Proc. of Workshop on Next Generation Software*, Rhodes, Greece, Apr. 2006.

[5] S. Friedman, P. Krishnamurthy, R. Chamberlain, R. K. Cytron, and J. E. Fritts. Dusty caches for reference counting garbage collection. In *Proc. of Workshop on Memory Performance: Dealing with Applications, Systems and Architecture*, St. Louis, MO, Sept. 2005.

[6] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of IEEE 4th Workshop on Workload Characterization*, 2001.

[7] J. L. Hennessey and D. A. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, 2003.

[8] R. Hough, P. Jones, S. Friedman, R. Chamberlain, J. Fritts, J. Lockwood, and R. Cytron. Cycle-accurate microarchitecture performance evaluation. In *Proc. of Workshop on Introspective Architecture*, Feb. 2006.

[9] P. Jones, S. Padmanabhan, D. Rymarz, J. Maschmeyer, D. V. Schuehler, J. W. Lockwood, and R. K. Cytron. Liquid architecture. In *Proc. of Workshop on Next Generation Software*, Santa Fe, NM, Apr. 2004.

[10] Free Hardware and Software Resources for System on Chip. `http://www.leox.org`.

[11] K. M. Lepak. *Exploring, Defining, and Exploiting Recent Store Value Locality*. PhD thesis, University of Wisconsin–Madison, 2003.

[12] K. M. Lepak and M. H. Lipasti. Silent stores for free. In *Proc. of 33rd ACM/IEEE Int'l Symp. on Microarchitecture*, pages 22–31, 2000.

[13] K. M. Lepak and M. H. Lipasti. Temporally silent stores. In *Proc. of 10th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 30–41, 2002.

[14] J. W. Lockwood. Evolvable Internet hardware platforms. In *Proc. of 3rd NASA/DoD Workshop on Evolvable Hardware*, pages 271–279, July 2001.

[15] S. Padmanabhan, R. K. Cytron, R. Chamberlain, and J. W. Lockwood. Automatic application-specific microarchitecture reconfiguration. In *Proc. of 13th Reconfigurable Architectures Workshop*, Rhodes, Greece, 2006.

[16] S. Padmanabhan, P. Jones, D. V. Schuehler, S. J. Friedman, P. Krishnamurthy, H. Zhang, R. Chamberlain, R. K. Cytron, J. Fritts, and J. W. Lockwood. Extracting and improving microarchitecture performance on reconfigurable architectures. *Int'l Journal of Parallel Programming*, 33(2–3):115–136, June 2005.

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

[18] μClinux – Embedded Linux/Microcontroller Project. `http://www.uClinux.org`.

[19] T. Wolf and M. A. Franklin. CommBench – A telecommunications benchmark for network processors. In *Proc. of IEEE Int'l Symp. on Performance Analysis of Systems and Software*, pages 154–162, Austin, TX, Apr. 2000.

[20] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. of 22nd Int'l Symp. on Computer Architecture*, pages 24–36, June 1995.

[21] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *ACM Computer Architecture News*, 23(1):20–24, Mar. 1995.