Quantifying Wasted Write Energy in the Memory Hierarchy

Charles Shelor, Jim Buchanan, and Krishna Kavi
Department of Computer Science and Engineering
University of North Texas
Denton, TX, USA

Ron Cytron
Department of Computer Science
Washington University
St. Louis, MO, USA

Abstract

Wasted writes occur when modified cache lines are evicted and written back to main memory even though the data contained in those lines is no longer needed by the program or does not change the existing memory contents. Wasted writes include values in retreating stacks and values in heap objects that have been deallocated. They also occur as a result of unmodified data values being written back as part of a modified cache line. Wasted writes consume energy, consume execution time as memory bandwidth and consume component lifetime of limited write-cycle technologies such as flash memory or phase-change memory (PCM). This paper characterizes the number and type of wasted writes through the memory hierarchy and quantifies the amount of potential energy savings that can be obtained from eliminating wasted writes. If all of the wasted writes could be eliminated from our benchmarks, 15.5% of the total memory subsystem energy could be saved.

1 Introduction

One of the major focuses in current research of computing systems is minimizing the power consumption of computations. This is a broad-based research theme as mobile computing devices strive for longer battery life while cloud-computing data centers and supercomputers are concerned with the massive power needs and cooling requirements for their systems. This is being addressed by the computing industry in many ways: continuing improvements in semiconductor technology, optimizing cache configurations, and low-power circuit-design rules are just a few examples. This paper proposes the reduction of wasted writes throughout the memory hierarchy as another method to reduce energy use within a computer system. The paper documents the type and quantity of wasted writes at each level of the memory hierarchy for various benchmarks. This information is then used to propose methods to reduce or eliminate the wasted writes at each of the levels.

The results from this research for a typical cache configuration show 59% of the bytes written to the L1 cache, 69% of the bytes written to the L2 cache, 66% of the bytes written to the L3 cache, and 44% of the bytes written to main memory fall into the wasted-writes categories. Eliminating all of these wasted writes would save 15.5% of the energy used by the cache-memory subsystem. Even if only 1/2 of the wasted writes are removed, nearly 8% of the cache-memory power use would be saved. This paper will show that wasted writes come from a variety of sources, and reducing these wasted writes requires cooperation between compiler and architectural enhancements for maximum benefits.

Section 2 classifies writes (or data modifications) for the purpose of identifying unnecessary writes. Section 3 describes the tools used and modifications to those tools required to perform the write classification and to collect the data for determining the wasted writes. Section 4 describes the experimental setup of benchmarks and the cache configurations used in the data-collection process. Section 5 discusses the results and analysis of the experiment. Section 6 proposes some methods of reducing the wasted writes. Section 7 describes the research to be performed to test and evaluate the proposed wasted-write reductions. Section 8 contrasts the work of this paper with other research projects with similar goals. Section 9 provides the conclusions derived from this research.

2 Write Classification

The team reviewed characteristics of write operations to determine a classification system. Each class was analyzed, and a determination was made as to whether the write was necessary for correct program execution or if it was a wasted write. The first three classes apply to processor writes and to cache-line write-back writes, while the last three classes apply only to cache-line write-backs. The following classes were developed and used for this paper:

Live Writes. A *live* write is when data is written to an address and changes its current value and later

that address is read by the application program. A *live* write is the common conception of all write accesses. A *live* write should never be eliminated as it would result in an incorrect program result. Thus, a *live* write is the only write category that is not a wasted write.

Useless Writes. A *useless* write is a processor write transaction that modifies the data at an address, but the changed data is never read by the application program. This may be the result of a subsequent write changing the data before it is read or by the application program terminating without reading the new data. As the information from a *useless* write is never used, the write can be eliminated without affecting correct program execution.

Dusty Writes. A *dusty* write is a processor write operation where the current data at the write address already matches the data being written. One example of this is a linked-list being followed back to its starting point. When this cache line is written back to memory, the write data matches the original data. Another example occurs in sorting routines where some items are already partially sorted and get written back to their original locations. *Dusty* writes are easy to detect during the write cycle by comparing the data being written to the existing data at that address.

Dead Writes. A *dead* write is a cache-line write operation where the address of the cache line is no longer active within the application program. One source of *dead* writes is when a heap block has been freed and there are dirty cache lines for that freed block that eventually get evicted from the cache and written back to the next level. Another source of *dead* writes is a retreating stack. Modified data left on the stack when a function returns will never be accessed again by the program.

Untouched Writes. An *untouched* write occurs when a cache line with its dirty bit set is written back and parts of the cache line have not been modified since the cache-line fill operation. Since the data has not been modified while it was in the cache line, it still matches the data in memory making this a wasted write. *Untouched* writes will not occur at the cache line or cache sub-block granularity as this condition implies the dirty bit for that data has not been set and a write-back would not be triggered.

Mixed Writes. A *mixed* write occurs when a cache line or cache sub-block contains a mixture of *live* writes with wasted writes. The necessary writes require the cache line or sub-block to be written to ensure program correctness, even if the majority of the cache line or sub-block consists of wasted writes. A *mixed* write that contains at least one *live* write byte cannot be considered a wasted write at cache-line granularity as the write must be performed for program correctness. However, cache lines that are a mixture

of only *dead*, *dusty*, *useless* and *untouched* writes are categorized as wasted writes.

3 Tools Used

The project chose to use Valgrind, Gleipnir, DineroIV and Cacti as the tools for this research.

Valgrind. Valgrind [10] was used to perform instrumented simulation of the benchmarks. Valgrind is a simulation framework allowing a variety of tools to monitor and interact with the program being simulated. There were no changes made to the core Valgrind operation.

Gleipnir. Gleipnir [7] is a data-structure analysis tool integrated into the Valgrind framework. Gleipnir was used to determine global, heap, or stack scope of the memory accesses and to generate the trace files of each memory access. The Gleipnir trace-output functions were modified to include the data values at each of the addresses in the trace as that information is required to detect *dusty* writes. Gleipnir was also modified to output a trace record for each change to the application stack pointer to detect *dead* writes from the unused portions of the stack. A final modification to Gleipnir added address and size information for all forms of *malloc()* and added address information for all *free()* function calls to the output trace file. This information was needed to detect *dead* writes to deallocated heap objects.

DineroIV. DineroIV [4] was used to simulate the cache activity from the Gleipnir trace files. The released form of DineroIV is data agnostic and performs all of its cache simulation using the trace addresses. DineroIV was modified to track data values to detect dusty writes. Dirty, valid, and last-access-type status bits were added for each cache-line byte to classify *live*, *useless*, and *untouched* accesses of each byte. Modifications were made to the logic to classify and count the different types of writes as they occurred throughout the memory hierarchy.

Cacti. Cacti [12] is a cache energy and access time estimation tool. Cacti version 6.0 was used to provide energy estimates for each level of the various cache configurations analyzed in the study. No modifications were made to the Cacti tool.

4 Experimental Setup

4.1 Benchmarks Analyzed

Five benchmarks totaling seven variations from the CPU2006 SPECmark series [11] were processed through Valgrind and Gleipnir. The SPEC benchmarks

selected for this study are representative of industry workloads and are sufficiently large to exercise the cache. Many of the smaller benchmarks, such as those in the MiBench [6] benchmark suite, were found to be components of applications rather than complete applications and would sometimes be wholly contained within the caches. In some cases the last-level cache was not even utilized in the benchmark's execution. The benchmarks selected for use in this study were required to have a minimum of 10 seconds and a maximum of 10 minutes of real-time execution. The minimum requirement assured the benchmark truly exercised the memory subsystem, while the maximum requirement is needed to create an upper bound on simulation time and trace-file size. Simulation execution times were as low as 37 minutes and as high as 52 hours for the selected benchmarks. The SPEC benchmarks used were bzip2, gcc_166, gcc_200, gcc_c-typeck, gobmk, hmmer, and mcf. The three variations of gcc were kept as they each had significantly different memory access profiles from the others.

4.2 Cache Configurations

There were fourteen cache configurations used to analyze wasted writes in this project. Small, nominal, and large 2-level caches without sub blocks; small, nominal, and large 3-level caches without sub blocks; nominal 3-level caches with 2, 4, and 8 sub blocks per cache line; nominal 3-level caches with 16, 32, 64, and 128 bytes per cache line; and nominal 3-level caches with an increasing number of bytes per cache line per level of 16/32/64, nominal-3L-mix1, and 32/64/128, nominal-3L-mix2. This variety of cache configurations allows determination of wasted write sensitivity to cache size, cache-line size, and use of sub blocks. Every cache configuration uses a split level-1 cache and a unified cache at all other levels. The nominal 2-level cache configuration is representative of caches similar to the Arm Cortex A-15 [1] cache with the shared L2 cache equally distributed among the cores (L1: 32K instruction, 32K data; L2: 1024K per core). The nominal 3-level cache configuration is representative of caches similar to the Intel Ivy Bridge [5] cache configuration with the shared L3 cache equally distributed among the cores (L1: 32K instruction, 32K data; L2: 256K unified; L3: 2048K per core). The small cache configurations are 1/2 the size of the nominal caches and represent caches either smaller than the nominal configuration or represent the effect of adding overhead functions for the operating system and its various processes to the benchmark task. The large cache configurations are 2 times the size of the nominal caches and can represent next-generation caches or a benchmark process getting a double allocation of the shared cache.

4.3 Energy Savings Estimation

A goal of the first phase of this project was an approximate potential energy savings if all wasted writes were eliminated. It is unlikely that all wasted writes can be removed, but this assumption establishes an upper bound on the savings that might be achieved. The assumption was made that reads and writes at each level required the same amount of energy. This is not true for PCM and flash-memory technologies where the write energy is significantly more than the read energy; however, for SRAM caches and DRAM memories this assumption is suitable. The Cacti cache-energy estimator was used to estimate the energy per access for each level of each cache configuration using 32 nm technology. The study produces results in terms of percentage of energy saved, so variations in technology and clock rate have minimal impact on the study results.

The energy required for memory-level accesses was computed by summing the energy needed to communicate between the CPU and the memory with the energy needed for the memory access. The transfer energy was computed using $E = 1/2\ V^2\ C$. C was chosen as 20 pF to represent a typical memory data signal's total capacitance for trace and connected devices. V was set to 1.5 Volts as the nominal voltage swing of single ended DDR3 memory devices. Address and control lines used 40 pF for C as there are more device loads on each address and control line. Each data line was toggled at 50% of the transfer rate based on the statistical assumption that each bit was 50% 0 and 50% 1. Thus, there is a 50% chance that the next bit is different from the present bit resulting in a 50% toggle rate being an appropriate value for the equation. Each address/control line was assumed to change once per cache-line access, based on typical DRAM memory burst-mode operation. The energy for the memory access was derived from the power required for a burst write (P = VDD * IDD) divided by the transition rate for the burst (E = P / T) multiplied by the number of transitions required to transfer a cache line on a 256-bit memory bus and multiplied by the number of memory chips needed to implement a 256-bit memory bus. The Micron MT41J512M8 [9] DDR3 memory device data sheet provided the VDD, IDD and T values for an 800 MHz memory subsystem. The energy for the cache to memory controller data transfer within the processor device and the energy for the memory-controller operation itself was assumed to be negligible for this phase of the research. As the final analysis is based on a percentage of energy that could be saved, moderate variations to these values should have little influence on the results.
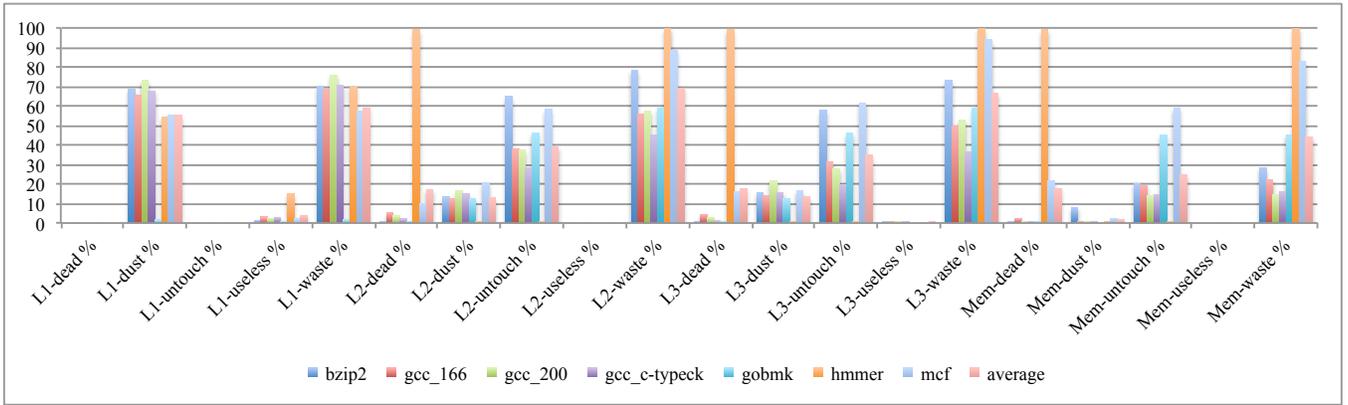
Figure 1: Wasted Write Breakdown by Level and Type for Nominal 3-Level Cache

5   Results and Analysis

All of our results are provided as percentages for each benchmark. This prevents longer-running benchmarks from dominating shorter-running benchmarks if access counts or actual energy values were used. All of the results shown in this paper are based on collecting data by individual bytes rather than application-level data objects as the present cache simulator does not maintain information about data-object size throughout the cache hierarchy. The multi-core cache simulator being developed for the next phase of this research will provide data-element size tracking. The memory trace files generated by Valgrind and Gleipnir for this research use virtual addressing. The authors believe that the difference between virtual and physical addressing will have minimal impact on this study, although physical addressing will be incorporated in the next phase of this research to validate this statement.

Figure 1 shows the wasted write breakdown for each level of the memory hierarchy for the nominal 3-level cache configuration. The x-axis labels, "LL-CAT %)", identify the measurement level in the memory hierarchy (L1 cache, L2 cache, L3 cache, or memory) and the wasted write category (dead, dusty, untouched, useless, or total-wasted). The y-axis indicates the percentage of bytes *written at that cache level* that belong to the indicated category. The number is computed by dividing the number of bytes written at that cache level in the indicated write category by the total number of bytes written at that cache level and expressing the result as a percentage. One observation that can be made is there are very few *useless* writes at any level and they have minimal impact to the total wasted writes of the system. Another observation is there are no *dead* writes and no *untouched* writes at the Level 1 cache. As stated earlier, if the processor is accessing memory, it cannot be classified as a *dead* write. Also

by definition, an *untouched* write can occur only during a cache-line write-back, so a processor-L1 transaction will never have an *untouched* write. A general trend can be observed where the percentage of *dusty* writes decreases as the level moves further from the processor. This is a result of the average time between writes at each cache level increasing as the level increases, reducing the chance of the same value being written multiple times. The *untouched* write category is generally the highest for each benchmark at the L2, L3 and memory levels, with the notable exception of hmmer with 99% *dead writes*. The hmmer benchmark has a large amount of heap activity with a very large memory footprint causing many cache-line evictions of deallocated heap objects, producing the very high *dead* writes beyond level 1 cache. This graph shows that no single type of wasted writes completely dominates all levels or all benchmarks; therefore all of the wasted write types should be addressed to maximize the possible savings. The nominal 3-level cache configuration, similar to the Intel Ivy Bridge, showed a benchmark average of 44.2% of all bytes written to memory as being wasted writes.

Figure 2 shows the total wasted write percentages by benchmark for all of the analyzed cache configurations measured at the memory level. The x-axis labels indicate the cache size as small, nominal, or large; indicate if it is a two-level cache, "-2L", or a three-level cache, "-3L"; indicate if there are subblocks in the cache line, "-2sb, -4sb, -8sb"; the cache-line block size if it is not the default 64 bytes, "-16blk, -32blk, -128blk"; and indicate if the configuration used a mixture of cache-line sizes, "-mix1" (16/32/64 bytes per cache line) or "-mix2" (32/64/128 bytes per cache line). The y-axis indicates the percentage of total wasted write bytes written to total bytes written at the DRAM memory level. This value is computed for a cache configuration by taking the total number of wasted write bytes written to memory and dividing it by the total number of
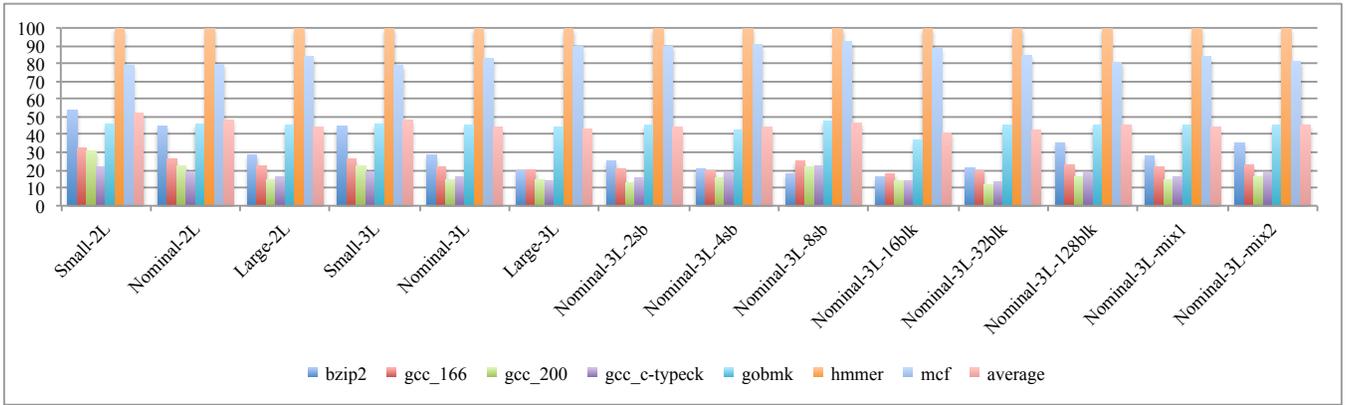
Figure 2: Wasted Write Breakdown by Cache Configuration at Memory Level

bytes written to memory for each benchmark and then averaging this percentage for the benchmarks. In general, smaller cache sizes are observed to have a slightly higher wasted write percentage than larger cache sizes. The 2-level caches had 52%, 48%, and 44% wasted writes for small, nominal, and large sizes respectively. The 3-level caches had 48%, 44%, and 43% wasted writes for small, nominal, and large sizes respectively. The higher rate of evictions of the smaller caches result in cache lines with a higher percentage of untouched bytes. However, in some cases, such as the mcf benchmark, the wasted write percentage increased slightly with increasing cache size. The moderate number of *dead* writes in mcf, 22% for nominal 3L, decreased with faster evictions with the smaller caches than *untouched* writes increased with the larger caches. Some small variations are seen among the caches with 2, 4, and 8 sub-blocks. However, the variations are minor with the 2 and 4 sub-blocks decreasing the average wasted writes by 0.02 and 0.18 percent. The cache configuration with 8 sub-blocks actually increased the percentage of wasted writes by 2.48%. The cause of the increase in wasted writes when a small decrease was expected has not yet been determined and will be further examined in the next phase of the research. The cache configurations with smaller block sizes resulted in smaller wasted writes, although not by a significant amount. The 16-byte block size yielded 41% wasted writes which is 3 percentage points less than the nominal 64-byte block size. The 128-byte block size yielded 46% or 1.4 percentage points higher than the nominal. This trend is expected as the larger cache-line sizes will likely contain larger amounts of untouched data. The cache configurations with a different block size per level yielded average wasted write percentages within 0.1 percentage point of the cache configuration with the matching L3 block size. Some runs were made with different set associativities (2, 4, 8 at L1 with 4, 8,

and 16 at L2 and L3), and they resulted in less than 0.5 percentage point variations. This data shows that cache size has a moderate effect on wasted write percentages, and all other cache configuration variations have negligible effects.

Similar information measured at the Level-1 cache, Level-2 cache, and Level-3 cache showed the same general trends, although some benchmarks have their peak value of wasted writes at different cache levels than others. This is simply a reflection of the differences in memory footprint and access sequences of the benchmarks.

The previous analyses have looked at each level of the memory hierarchy independently and displayed the results as percentages at that level. The analysis of total energy savings must be computed for the total memory subsystem before being made a percentage as the energy per access at each level differs and the frequency of access at each level is different. The total energy used at each level was computed by multiplying the energy required per access at that level times the sum of the instruction accesses, plus the sum of all read accesses, plus the sum of all write accesses. The wasted energy at each level was computed by multiplying the energy required per access at that level times the total wasted accesses at that level. The total energy and wasted energy of each level were summed to get the total energy and wasted energy of the memory subsystem. Taking the wasted energy of the subsystem and dividing it by the total energy of the subsystem generated the wasted energy percentages shown in Figure 3. The x-axis labels show the same cache configurations used in and described for Figure 2. The y-axis shows the percentage of energy wasted by benchmark within each cache configuration. The wasted write energy savings range from 13.2% for the large 3-level cache to 19.6% for the nominal 3-level cache using 8 sub-blocks. The Nominal-3L cache configuration shows a
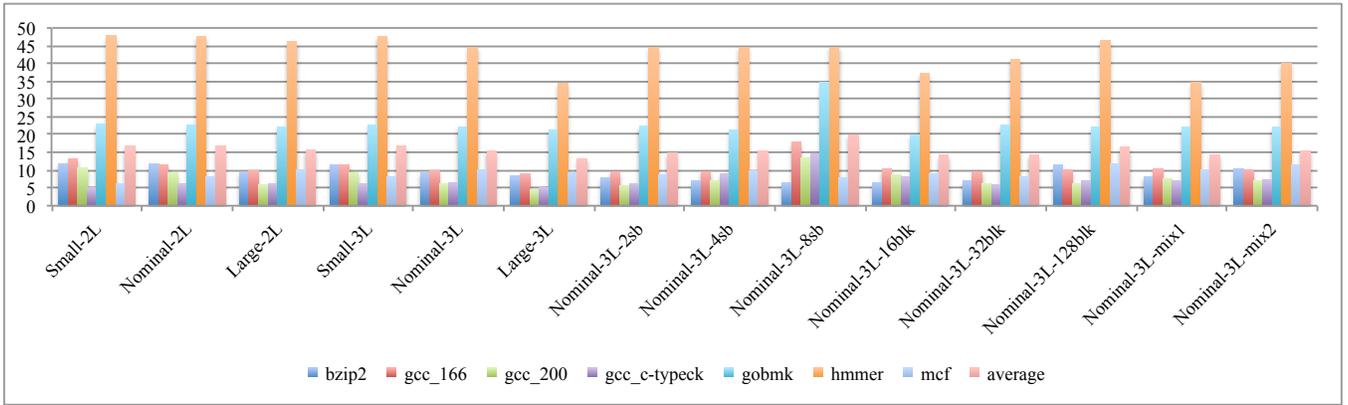
Figure 3: Wasted Energy by Cache Configuration and Benchmark

minimum wasted energy savings of 6.1% for the gcc_200 benchmark, a maximum wasted energy of 44% for the hmmer benchmark and an average wasted energy of 15.5%. The wasted write energy percentages for flash technology and PCM technology will be higher than those for DRAM systems because the energy required for writing in those technologies is significantly higher than the energy used for reading.

The potential energy savings must be summarized at the memory subsystem level because of interesting interactions between the cache levels. The Cacti energy estimates for the nominal 3-level cache are 0.16 nJ, 0.03 nJ, and 0.11 nJ for the L1, L2, and L3 accesses respectively. The L2 is lower energy than the L1 as it is slower and only slightly larger. The L3 energy is higher than the L2 because it is much larger. A cache-line write to DRAM requires 16.11 nJ. Since accessing memory requires 100 times the energy of accessing the L1 cache, the results might be skewed to the wasted write percentages seen in Figure 2 for the memory level. However, approximately 99% of all accesses are handled by the cache such that the actual number of memory transactions is much less than the number of cache transactions. There are 100 times more accesses to cache than memory, but each access to memory requires 100 times the energy as a cache access. The true picture of energy use is obtained only when all of the memory hierarchy is included.

6 Proposed Implementations

This paper has identified four types of wasted writes through the memory hierarchy. This section addresses methods that can reduce or eliminate the wasted writes.

A *useless* write is a write whose value is not read in the future. There is no feasible method for the memory

subsystem to determine at the time of the write whether or not the data value is going to be read in the future. However, Butts [3] uses a prediction mechanism in the processor pipeline for *useless* instruction elimination that successfully eliminates 79% of *useless* instructions. There was a reduction in register writes of 1.7% to 11.3% in the benchmarks analyzed by Butts which is similar to the 1.6% to 13.6% of useless writes for the level 1 cache shown in Figure 1. Figure 1 also shows that the *useless* write category has a very small contribution to wasted writes for the level 2 cache, the level 3 cache, and the main memory. This indicates that attempting further reduction of useless writes at those memory levels will be both difficult and have little benefit. However, since 99% of memory accesses are handled by the level 1 cache, minimizing the *useless* writes at the level 1 cache might have a noticeable improvement in the cache energy savings. As shown in Butts, many of the *useless* instructions were created by instruction scheduling by the compiler. Therefore, it might be possible for a compiler liveness analysis to determine that particular writes are *useless* and to remove them through compiler optimizations and different instruction scheduling algorithms.

Dead writes can be minimized with fairly simple architectural changes and run-time library updates. Marking a cache line as *invalid* or making a cache line *clean* by clearing the dirty bit are common cache operations used when terminating a program. This prevents writing stale data from the terminated process to memory that may have been reallocated to a subsequent process. The addition of *cache-line-invalidate* or *cache-line-clean* operations to the run-time library functions that free allocated memory will eliminate *dead* writes when heap objects are deallocated. This approach can be implemented with existing cache systems. Another approach could be implemented within the cache with a *cache-line-batch-invalidate* operation that accepts an

address argument and a size argument. This operation would invalidate all of the cache lines associated with the given block of memory with a lower processor overhead than performing the invalidate one line at a time. Additionally, It may be useful to have all run-time memory allocations aligned to cache-line boundaries to ensure that no two heap objects can share a single cache line.

Dusty writes can be detected in run-time hardware by comparing the written data to the existing data. When the values are equal, the dirty bit of the destination cache line would be left unchanged rather than being set. (If it is already set, it must remain set.) If all of the writes to that cache line are *dusty* writes, then the line will remain clean and will not have to be written back to the next level when evicted. Additionally, there may be compiler analyses that can detect *dusty* write conditions and remove them during optimization or group them to share common cache lines.

Untouched writes can be minimized by compiler optimizations that group variables that are written at similar times together. For example, if there are 8 variables that are written from a short code fragment, the compiler can detect this and allocate addresses such that the variables share a single cache line rather than being in 8 separate cache lines. Not only will there be fewer untouched bytes in the one cache line that was modified, there is only 1 dirty cache line created by that code fragment where it could have potentially created 8 dirty cache lines. This could be as simple as rearranging the order of elements within a structure in some programs.

7 Future Work

There are several tasks to be performed in the next phase of this project. In addition to the new work, the authors will add more benchmarks to the analysis to broaden the application base of this effort. The new benchmarks will still comply with the real-time execution minimum and maximum limits to ensure they exercise the memory subsystem sufficiently, yet remain within reasonable simulation execution times.

The current project collected data by bytes, sub-blocks, and blocks. A fourth category will be implemented to track at the program variable instance. This will eliminate the upper parts of pointer and index values from being marked as dusty writes. This will also assist in detecting when compiler optimizations in structure alignment and variable grouping have been effective. Tracking data by program instance will require further modifications to the cache simulator to track data-element sizes within the cache lines.

Simulations using physical addresses, multiple cores executing simultaneously, shared memory, and shared caches will be used in the next phase to more closely model actual system performance. This will require development of a multi-core cache simulator.

The list of possible solutions will be expanded in the next phase. Compiler optimizations will be implemented when possible or emulated by address manipulation within the trace file when appropriate. Architectural features will be implemented and simulated. Each of these optimizations will be used to create new trace files that will be analyzed to determine the amount of wasted writes that were eliminated by the optimization. Each architectural feature will be assessed to determine its costs with respect to increasing silicon area, increases in energy use, and impacts on critical paths. This will provide a cost-benefit rating for each of the methods of wasted write reduction.

8 Related Work

Bock [2] analyzed the impact of wasted write-backs on the endurance and energy use of PCM main memory. Although the specific tools varied, the Bock analysis framework was very similar to that used in this paper. The main difference in this paper was attacking the more general problem of wasted writes at each level of the hierarchy and determining potential energy savings in DRAM memories. This paper also used the Cacti cache energy estimation tool to provide energy estimates at each level of the cache as the energy per access and number of accesses at each level vary dramatically. This allowed us to compute the potential energy savings for the entire memory subsystem rather than just the memory level and to determine that the potential energy savings would be worth pursuing in the next phase.

Lepak [8] analyzed "silent stores" showing an 11% performance improvement achieved by detecting and eliminating these stores in a two-level write-through system. These "silent stores" correspond to our *dusty* write category that are shown in Figure 1 to be the dominant wasted write at the L1 cache, a minor contributor to the wasted writes at the L2 and L3 cache, and almost negligible at the main-memory level of the write-back cache used in this study. The Lepak paper considered microarchitecture changes in the pipeline and changes to Error Correction Codes (ECC) as methods to implement their silent store reductions while our focus is compiler optimizations and cache implementations.

Butts [3] analyzed the detection and elimination of dynamic dead-instructions with a mechanism similar to branch prediction and eliminates execution of those instructions whose results are not used in subsequent

code. Their work eliminates generation of values in registers in addition to write cycles from store instructions. These stores would correspond to the *useless* write category of wasted writes discussed in our paper. We saw minimal *useless* writes beyond the level 1 cache in Figure 1. However, most of the benefits of dynamic dead-instruction elimination occur within the execution pipeline and are therefore complementary and additive with respect to our paper.

9 Conclusion

We have characterized and quantified the wasted writes throughout the cache-memory hierarchy using industry-standard benchmarks and shown significant amounts of wasted writes occur at each level of the hierarchy. We have shown that elimination of all of the wasted writes would save 15% of the power consumed in a typical cache-memory subsystem and have embarked on future work to determine how much of the power savings can be achieved through compiler and architectural enhancements. Some of these enhancements will reduce cache conflicts producing higher cache hit rates with a subsequent reduction in read power and reduction in memory bandwidth. Additionally, the reduction in wasted writes will extend the lifetime of memory technologies that have limited write-cycle endurance. Reduction in wasted writes can also be affected by programming practices. For example, a programmer who does not free memory when it is no longer needed will prevent the system from classifying those writes as dead writes.

References

[1] Arm Cortex-A15 Technical Reference Manual, Chapters 6-7, http://infocenter.arm.com/help/ index.jsp?topic=/com.arm.doc.ddi0438g/index.html

[2] Bock, S., Childers, B., Melhem, R., Mosse, D., Zhang, Y. Analyzing the Impact of Useless Write-Backs on the Endurance and Energy Consumption of PCM Main Memory. ISPASS 2011, pp 56-65, IEEE Conference Publications, NY, NY (2011)

[3] Butts, J. A., Sohi, G. Dynamic dead-instruction detection and elimination. Proceedings of the 10th international conference on Architectural support for programming languages and operating systems (ASPLOS-X), pp 199-210, ACM, New York, NY, USA, (2002)

[4] DineroIV web site, http://pages.cs.wisc.edu/~markhill/DineroIV/

[5] Intel Core i7-3770K Ivy Bridge Processor Review, http://hothardware.com/Reviews/Intel-Core-i73770K-Ivy-Bridge-Processor-Review/?page=3

[6] Iqbal, S. M. Z., Liang, Y., Grahn, H., ParMiBench - An Open-Source Benchmark for Embedded Multiprocessor Systems. In IEEE Computer Architecture Letters, Vol 9, No. 2, July-December 2010, IEEE Computer Society, New York

[7] Janjusic, T., Kavi, K., Potter, B., Gleipnir: A Memory Analysis Tool. International Conference on Computational Science 2011, pp 2058-2067, Elsevier Ltd.

[8] Lepak, K. M., Lipasti, M. H., Silent Stores for Free. In IEEE/ACM International Symposium on Microarchitecture, December 2000, MICRO 33, pp 22-31, IEEE Computer Society, New York

[9] MT41J512M8 DDR3 SDRAM Data Sheet, Micron Technology, http://www.micron.com (2009)

[10] Nethercote, N., Seward, J. Valgrind: A Program Supervision Framework, In: Electronic Notes in Theoretical Computer Science 89 No 2. pp 44-66. Elsevier Science B. V. (2003)

[11] SPEC Benchmark Information, http://www.spec.org/cpu2006/Docs

[12] Thoziyoor, S., Ahn, J. H., Monchiero, M., Brockman, J. B., Jouppi, N. P., A Comprehensive Memory Modeling Tool and its Application to the Design and Analysis of Future Memory Hierarchies. International Symposium on Computer Architecture 2008, pp 51-62, IEEE Press, New York