Potential Energy Savings Through Eliminating Unnecessary Writes in the Cache-Memory Hierarchy

Charles Shelor[*], Jim Buchanan[*], and Krishna Kavi[*]
University of North Texas, Denton, TX USA

Ron Cytron[†]
Washington University, St. Louis, MO USA

Abstract

Many different areas of research are addressing reduction of energy in computing systems. These include new semiconductor technologies, low-power design rules, clock gating, and voltage/frequency scaling, just to name a few. This paper presents a new energy reduction approach involving the compiler and computer system architecture. Eliminating unnecessary writes in the system has the potential to reduce energy by 15 percent in the cache-memory hierarchy. Unnecessary writes occur when modified cache lines are evicted and written back to the next cache level or to main memory even though the modified data contained in those lines is no longer needed by the program or does not change the existing memory contents. Unnecessary writes include values in retreating stacks and values in heap objects that have been deallocated. Unnecessary writes also occur as a result of unmodified data values being written back as part of a modified cache line. Unnecessary writes affect a computer system's performance in several ways. The energy used by these writes is wasted energy. The unnecessary writes require execution time as memory bandwidth and they reduce the component lifetime of limited write-cycle technologies such as flash memory or phase-change memory (PCM). This paper characterizes the number and type of unnecessary writes through the memory hierarchy and quantifies the amount of potential energy savings that can be obtained from eliminating unnecessary writes.

Key Words: Computer architecture, energy reduction, compiler optimizations, memory systems, phase change memory.

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, low-power circuit-design rules, clock gating, optimizing cache configurations, and voltage-frequency scaling are just a few examples. This paper proposes the reduction of unnecessary writes throughout the memory hierarchy as another method to reduce energy use within a computer system. The paper documents the type and quantity of unnecessary writes at each level of the memory hierarchy for various benchmarks. This information is then used to propose methods to reduce or eliminate the unnecessary writes at each of the levels.

The results from this research for a typical cache configuration show 59 percent of the bytes written to the L1 cache, 69 percent of the bytes written to the L2 cache, 66 percent of the bytes written to the L3 cache, and 44 percent of the bytes written to main memory fall into the unnecessary-writes categories. This research also shows that cache size does have a moderate effect on the unnecessary write percentages, while other cache configuration aspects have negligible effects. Eliminating all of these unnecessary writes would save 15.5 percent of the energy used by the cache-memory subsystem. Even if only 1/2 of the unnecessary writes were removed, nearly 8 percent of the cache-memory power use would be saved. This paper shows that unnecessary writes come from a variety of sources, and reducing these unnecessary writes requires coordinated compiler and architectural enhancements for maximum benefits.

The rest of the paper is organized as follows. 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 unnecessary 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 implementations for reducing the unnecessary writes. Section 7 describes the research to be performed to test and evaluate the proposed unnecessary-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 basic premise of an unnecessary write is that the

[*] Department of Computer Science and Engineering.
[†] Department of Computer Science.

elimination of that write-operation would not cause incorrect program behavior. Thus, the functional behavior of a program with the unnecessary writes eliminated would be indistinguishable from the functional behavior of the program with the unnecessary writes left intact. Write operations within the memory hierarchy were analyzed to determine a classification system. A total of six classes of write operations were established based on the characteristics of the write with respect to the active software program and the computer system. The first three classes apply to both processor writes and to cache-line write-back writes. The last three classes apply only to cache-line write-backs.

2.1 Live Write

A *live* write is when data is written to an address, changes the current value at that address, and that address is later read by the application program to retrieve the changed value. A *live* write is the common conception of all write accesses. A *live* write should never be eliminated, as it would result in incorrect program execution as the subsequent read would retrieve the old, incorrect data. A *live* write is the only write class that is not an unnecessary write.

2.2 Useless Write

A useless write is a write transaction that modifies the data at an address, but the application program never reads the changed data. 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 making a useless write an unnecessary write.

2.3 Dusty Write

A *dusty* write is a 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. Another source of *dusty* write bytes are pointers and counters where the upper data bytes change infrequently relative to the lower data bytes. *Dusty* writes are easy to detect during the write cycle by comparing the data being written to the existing data at that address. Since a *dusty* write does not change data at the written address it is an unnecessary write.

2.4 Dead Write

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 the program has freed a heap block of memory and there are dirty cache lines for that freed block. Those cache lines eventually get evicted from the cache and written back to the next cache level or memory. Another source of *dead* writes is a retreating stack. Modified data that is left on the stack when a function returns will not be accessed again by the program. As the data written during a *dead* write is no longer valid for the application program, the write is unnecessary.

2.5 Untouched Write

An *untouched* write occurs when only a part of a cache line is accessed and modified. This happens because cache lines are larger than data objects. However, the cache keeps dirty bits on a cache line basis and cannot distinguish untouched portions of the line from modified portions. Thus when the cache line is written back, both touched and untouched portions will be written back. The *untouched* bytes are unnecessary writes as they are the same value that was originally read from memory.

2.6 Mixed Write

A *mixed* write occurs when a cache line or cache sub-block contains more than one write type. A *mixed* write that contains at least one *live* write byte cannot be considered an unnecessary 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 unnecessary writes for the cache line.

3 Tools Used

The project chose to use Valgrind, Gleipnir, DineroIV and Cacti as the tools for this research. These tools are all public domain tools and Valgrind, DineroIV, and Cacti are widely used in computer architecture research.

Valgrind. Valgrind [10] was used to perform instrumented simulation of the applications. 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 for our research.

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 file of the memory accesses. For the purpose of this research, 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 as that information is required to detect *dead* writes from a retreating 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. An example of Gleipnir output is shown in Figure 1. Each line begins with a “type code” letter indicating the type of memory transaction or system operation.

Type	Pid	Address	Size	Data	Scope
A	8407	054f87b0	242		
F	8407	054f87e0			
I	8407	004c2b4a9	4	4883c440	
L	8407	0007143c0	4	000001c3	G
M	8407	7ff000158	8	0049ae7a00000000	S
P	8407	7ff000110			
S	8407	0054ea24c	4	000002bd	H
#	This entire line is a comment				

Figure 1: Example of Gleipnir trace output

The type codes are defined as:

- A Memory allocation operation
- F Memory free operation
- I Instruction fetch
- L Load data from memory
- M Modify memory location
- P Stack pointer change
- S Store data to memory
- # Comment line

The next field is the decimal process identification number for multiple core trace files and is required for all transaction types except comments. The next field is the hexadecimal address of the memory transaction or operation and is required for all transaction types except comments. The next field is the decimal number of bytes for the memory transaction or operation. This field is unused for ‘F’ and ‘P’ trace lines. The next field is the hexadecimal data for ‘I’, ‘L’, ‘M’, and ‘S’ trace lines. The final field is the scope of the memory where ‘G’ represents global scope, ‘H’ represents heap memory access, and ‘S’ indicates a stack access. This is required on ‘L’, ‘M’, and ‘S’ trace lines.

DineroIV. DineroIV [5] 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. For the purpose of this research, 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. Output functions were added to produce cache simulation statistics files with the counts of each type of memory access by cache level and scope. These files included a header specifying the benchmark being run and the cache configuration name. Then a separate section for each cache begins with the cache name and its parameters followed by a list of memory access counts by memory area, access type, bytes transferred, sub-blocks transferred, and cache lines transferred. Figure 2 shows the level 2 cache statistics for the gcc_166 benchmark. This shows the level 2 caches is unified, 512 Kbytes total size, a cache line of 64 bytes, a sub-block of 64 bytes, 8-way associativity, and 1024 sets. The statistics files are input to a data reduction program that merges the different benchmark data results to compute the energy consumption and compare the results by benchmark or cache configuration.

```
gcc_166,Large-3L
l2-ucache,512,64,64,8,1024
Global,reads,5483712,85683,85683,
Global,untch,322516,0,0,
Global,lives,326075,6,6,
Global,usels,0,0,0,
Global,dusts,101297,2572,2572,
Global,deads,0,0,0,
Global,mixed,0,9139,9139,
Heap,reads,34290560,535790,535790,
Heap,untch,18006417,0,0,
Heap,lives,24557859,4984,4984,
Heap,usels,0,0,0,
Heap,dusts,5940405,18396,18396,
Heap,deads,1715351,28630,28630,
Heap,mixed,0,732678,732678,
Stack,reads,1038272,16223,16223,
Stack,untch,234988,0,0,
Stack,lives,111923,17,17,
Stack,usels,0,0,0,
Stack,dusts,211201,124,124,
Stack,deads,426976,9180,9180,
Stack,mixed,0,6071,6071,
Instr,reads,77214272,1206473,1206473,
```

Figure 2: Example output from a DineroIV cache simulation

Cacti. Cacti [13] 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 [12] 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 or kernels 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 1 second 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 each other.

4.2 Cache Configurations

There were fourteen cache configurations used to analyze unnecessary 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 is used to determine if the amount of unnecessary writes is sensitive to any particular cache configuration parameter or specific parameter combinations. Every cache configuration uses a split level-1 (instruction and data) 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 [4] 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 unnecessary writes were eliminated. It is unlikely that all unnecessary 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 validity of 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 the standard energy equation for switching an electronic signal: $E = 1/2\ V^2\ C$. The value of C for data lines was chosen as 20 pF to represent a typical memory data signal's total capacitance for the PCB trace capacitance and capacitance of the connected memory devices. The value of C for address and control lines was set to 40 pF as there are more memory devices on each address and control signal. V was set to 1.5 Volts as the nominal voltage swing of single ended DDR3 memory devices. Each data line was toggled at 50 percent of the transfer rate based on the statistical assumption that each bit was 50 percent 0 and 50 percent 1. Thus, there is a 50 percent chance that the next bit is different from the present bit resulting in a 50 percent 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.

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 3 shows the unnecessary write breakdown for each level of the memory hierarchy for the nominal 3-level cache configuration. The *x*-axis labels, "*LL-category*%)'", identify the measurement level in the memory hierarchy (*LL*: L1 cache, L2 cache, L3 cache, or memory) and the unnecessary write category (*category*: dead, dusty, untouched, useless, or total-wasted, where total-wasted is the total percentage of all unnecessary writes). The *y*-axis indicates the percentage of bytes *written at the indicated 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 unnecessary 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 percent *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 unnecessary write completely dominates all levels or all benchmarks; therefore all of the unnecessary 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 percent of all bytes written to memory as being unnecessary writes.

Figure 4 shows the total unnecessary write percentages by benchmark for all of the analyzed cache configurations measured at the memory level. The *x*-axis labels indicate the

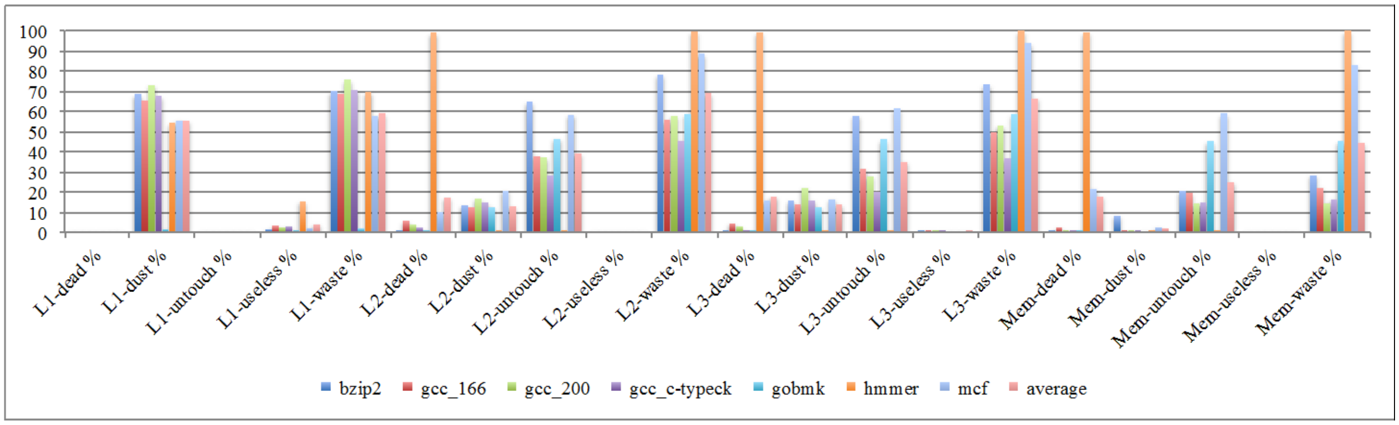

Figure 3: Unnecessary write breakdown by level and type for nominal 3-level cache

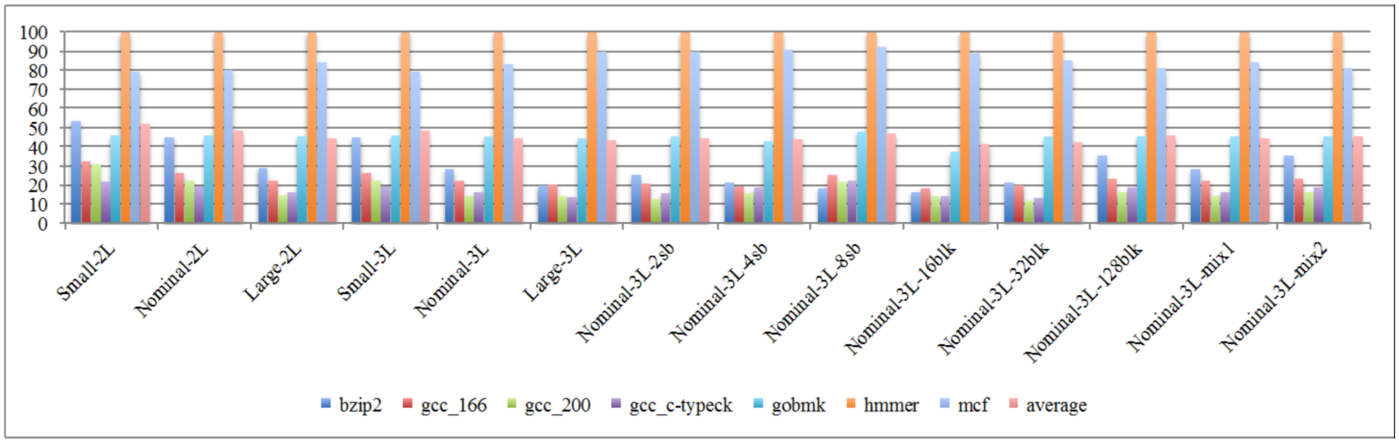

Figure 4: Unnecessary Write Breakdown by Cache Configuration

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 sub-blocks 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-unnecessary-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 unnecessary-write-bytes written to memory and dividing it by the total number of 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 unnecessary write percentage than larger cache sizes. The 2-level caches had 52 percent, 48 percent, and 44 percent unnecessary writes for small, nominal, and large sizes, respectively. The 3-level caches had 48 percent, 44 percent, and 43 percent unnecessary 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 unnecessary write percentage increased slightly with increasing cache size. The moderation number of *dead* writes in mcf, 22 percent for nominal 3L, decreased with faster evictions in 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 unnecessary writes by 0.02 and 0.18 percent. The cache configuration with 8 sub-blocks actually increased the percentage of unnecessary writes by 2.48 percent. The cause of the increase in unnecessary 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 unnecessary writes, although not by a significant amount. The 16-byte block size yielded 41 percent unnecessary writes which is 3 percentage points less than the nominal 64-byte block size. The 128-byte block size yielded 46 percent or 1.4 percentage points higher than the nominal. This trend was 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 unnecessary write percentages within 0.1 percentage point of the cache configuration with the matching L3 block size. Some runs were made with different set associativity (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 unnecessary 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 unnecessary 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 by the sum of the instruction accesses, plus the sum of all read accesses, plus the sum of all write accesses. The potential energy savings at each level was computed by multiplying the energy required per access at that level by the total number of unnecessary accesses at that level. The total energy and potential energy savings of each level were summed to get the total energy and potential energy savings of the complete memory subsystem. Taking the potential energy savings of the subsystem and dividing it by the total energy of the subsystem generated the potential energy savings percentages shown in Figure 5. The *x*-axis labels show the same cache configurations used in and described for Figure 4. The y-axis shows the percentage of potential energy savings by benchmark within each cache configuration. The potential

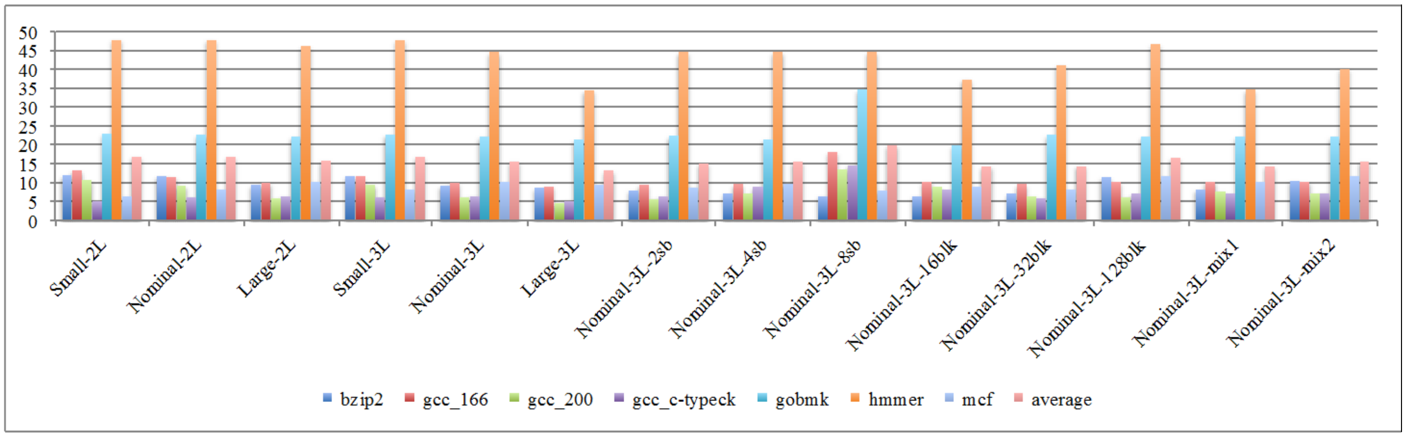

Figure 5: Potential energy savings by cache configuration and benchmark

energy savings range from 13.2 percent for the large 3-level cache to 19.6 percent for the nominal 3-level cache using 8 sub-blocks. The Nominal-3L cache configuration shows a minimum potential energy savings of 6.1 percent for the gcc_200 benchmark, a maximum potential energy savings of 44 percent for the hmmer benchmark and an average potential energy savings of 15.5 percent. The potential energy savings 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 unnecessary write percentages seen in Figure 4 for the memory level. However, the cache handles approximately 99 percent of all accesses 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 unnecessary writes through the memory hierarchy. This section addresses methods that can reduce or eliminate the unnecessary writes. *Untouched* writes are the largest unnecessary write contribution at all levels past level 1. One approach to reducing *untouched* writes is to only write the changed data in the cache line. However, this actually has a negligible energy savings as most of the energy in an access is accessing the row in the memory array. Blocking the write operation for unchanged data will not save any energy. *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 those variables share a single cache line rather than being in up to 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 changing the assigned address to some scalar variables or rearranging the order of elements within a structure in some programs. Reducing the number of dirty lines and write-backs will improve the cache-hit efficiency resulting in a slight improvement in program execution time.

Dead writes are the second largest contributor to unnecessary writes in the L2 and subsequent levels of memory. *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. These solutions need to be analyzed to determine the hardware cost, the energy cost, and the runtime performance cost to provide a cost/benefit analysis of the features.

Dusty writes are the largest contributor to unnecessary writes at the L1 cache level. *Dusty* writes can be detected in run-time hardware by comparing the written data to the existing data. By the time the *dusty* write is recognized, it is too late to prevent access to the cache line, so no energy can be saved at the present write cycle. However, when the values are equal, the dirty bit of the destination cache line can 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.

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 percent of *useless* instructions. There was a reduction in register writes of 1.7 to 11.3 percent in the benchmarks analyzed by Butts which is similar to the 1.6 to 13.6 percent of *useless* writes for the level 1 cache shown in Figure 3. Figure 3 also shows that the *useless* write category has a very small contribution to unnecessary 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 the level 1 cache handles 99 percent of memory accesses, 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.

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 write accesses 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 of present day processors. This requires development of a new 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 unnecessary 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 unnecessary write reduction.

8 Related Work

Bock [2] analyzed the impact of unnecessary 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 unnecessary 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 of this research.

Lepak [8] analyzed "silent stores" showing an 11 percent 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 3 to be the dominant unnecessary write at the L1 cache, a minor contributor to the unnecessary 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 unnecessary writes discussed in our paper. We saw minimal *useless* writes beyond the level 1 cache in Figure 3. 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.

Shidal [11] is also looking at ways to more efficiently utilize caches by reducing write-backs from cache due to objects that will be removed through garbage collection. Our research is currently limited to languages that explicitly free memory, while Shidal's work is oriented to languages with background garbage collection. A final solution to unnecessary writes will benefit from merging the results of both activities, rather than selecting one of the two approaches.

9 Conclusion

We have characterized and quantified the unnecessary writes throughout the cache-memory hierarchy using industry-standard benchmarks and shown significant amounts of unnecessary writes occur at each level of the hierarchy. We have shown that cache size has a moderate effect on the amounts of unnecessary writes and that other cache configuration parameters have negligible effects on the amount of unnecessary writes. We have shown that elimination of all of the unnecessary writes would save 15 percent 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 unnecessary writes will extend the lifetime of memory technologies that have limited write-cycle endurance. Reduction in unnecessary 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.

Acknowledgements

This research is supported in part by NSF award #1237417 and by industrial memberships of the NSF Net-Centric IUCRC. The authors wish to acknowledge the support given by Tomislav Janjusic of UNT on Valgrind and Gleipnir usage and Mike Ignatowski and Dave Mayhew of AMD for their suggestions and insights into cache-memory subsystems of current and future computer systems.

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, 2012.

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

[3] J. A. Butts and G. Sohi, "Dynamic Dead-Instruction Detection and Elimination," *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-X), ACM, New York, NY, USA, pp. 199-210, 2002.

[4] M. Chiappetta, "Intel Core i7-3770K Ivy Bridge Processor Review," http://hothardware.com/Reviews/Intel-Core-i73770K-Ivy-Bridge-Processor-Review/?page=3, April 23, 2012.

[5] M. Hill, DineroIV web site, http://pages.cs.wisc.edu/~markhill/DineroIV/, 1998.

[6] S. M. Z. Iqbal, Y. Liang, and H. Grahn, "ParMiBench - An Open-Source Benchmark for Embedded Multiprocessor Systems," IEEE Computer Architecture Letters, IEEE Computer Society, New York, 9(2):45-48, July-December 2010.

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

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

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

[10] N. Nethercote and J. Seward, "Valgrind: A Program Supervision Framework," Electronic Notes in Theoretical Computer Science, Elsevier Science B. V., 89(2):44-66, 2003.

[11] J. Shidal, Z. Gottlieb, R. Cytron, and K. Kavi, "Trash in Cache: Detecting Eternally Silent Stores," *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC-2014)*, Edinburgh, Scotland, Co-located with PLDI 2014, June 13, 2014.

[12] SPEC Benchmark Information, http://www.spec.org/cpu2006/Docs, 2006

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

Charles Shelor is currently a PhD candidate in Computer Engineering at the University of North Texas with research interests in computer architecture, dataflow processing, and embedded systems. He received an MS in Electrical Engineering from Louisiana Tech University in 1976 and BS in Electrical Engineering, Magna Cum Laude, from Louisiana Tech University in 1975. He has been a registered professional engineer since 1982 and is a member of IEEE and ACM.

He has been granted 13 patents through 25 years of embedded systems design with ASIC and FPGA development experience with Lockheed Martin, Efficient Networks and Alcatel-Lucent plus 6 years of processor design with Cyrix and Via Technologies. He also was an ASIC and FPGA design methodology consultant teaching structured, reusable design techniques for 4 years as Shelor Engineering. He was the author of the "VHDL Designer" feature for the "VHDL International Times" newsletter for 2 years. He won the VHDL International programming contest in the industry category in 1994.

Jim Buchanan received the B.S. degree in Computer Engineering from the University of North Texas (UNT), Denton, TX, USA, in 2013. He is currently pursuing the Ph.D. degree in Computer Science and Engineering at UNT. Jim also received the B.S. degree in Kinesiology from Texas A&M University, College Station, TX, USA, in 2000.

He spent a decade in sport marketing and the golf industry prior to his time at UNT. He is currently the Engineering Labs Technician for Computer Science and Engineering overseeing the embedded systems, communications systems, and senior design labs. His research interests involve computer architecture and sustainability of embedded and real-time systems.

Krishna Kavi is currently a Professor of Computer Science and Engineering at the University of North Texas (UNT) and the Director of the National Science Foundation (NSF) Industry/University Cooperative Research Center for Net-Centric and Cloud Software and Systems. During 2001-2009, he served as the Chair of CSE at UNT. He also held an Endowed Chair Professorship in Computer Engineering at the University of Alabama in Huntsville, and served on the faculty of the University Texas at Arlington. He was a Scientific Program Manger at NSF during 1993-1995. He served on several editorial boards and program committees.

His research is primarily on Computer Systems Architecture including multi-threaded and multi-core processors, cache memories, 3D DRAMs and Processing in Memory. He also conducted research in the area of software engineering, parallel processing, and real-time systems. He published more than 200 technical papers in these areas. He received his PhD from Southern Methodist University in Dallas Texas and a BS in EE from the Indian Institute of Science in Bangalore, India.

Ron K. Cytron is a Professor of Computer Science and Engineering at Washington University. His research interests include optimized middleware for embedded and real-time systems, fast searching of unstructured data, hardware/runtime support for object-oriented languages, and computational political science.

Ron has over 100 publications and 10 patents. He has received the SIGPLAN Distinguished Service Award and is a co-recipient of SIGPLAN Programming Languages Achievement Award. He served as Editor-in-Chief of ACM Transactions on Programming Languages and Systems for 6 years. He participated in writing the Computer Science GRE Subject Test for 8 years and chaired the effort for 3 years.

He is a Fellow of the ACM.