Analyzing the Impact of Useless Write-Backs on the Endurance and Energy Consumption of PCM Main Memory

Santiago Bock, Bruce Childers, Rami Melhem, Daniel Mossé, Youtao Zhang
Department of Computer Science
University of Pittsburgh
{sab104, childers, melhem, mosse, zhangyt}@cs.pitt.edu

***Abstract*—Phase Change Memory (PCM) is an emerging technology that has been recently considered as a cost-effective and energy-efficient alternative to traditional DRAM main memory. Due to the high energy consumption of writes and limited number of write cycles, reducing the number of writes to PCM can result in considerable energy savings and endurance improvement. In this paper, we introduce the concept of *useless write-backs*, which occur when a dirty cache line that belongs to a dead memory region is evicted from the cache (a dead region is a memory location that is not used again by a program). Since the evicted data is not used again, the write-back can be safely avoided to improve endurance and energy consumption.**

This paper presents a limit study on the improvement that passing information to the memory system about useless write-backs has on the endurance and energy consumption of systems based on PCM main memory. We developed algorithms to measure the number of useless write-backs to PCM for three different types of memory regions and we present an energy model to determine the maximum energy savings that could potentially be achieved through such a scheme. Our results show that avoiding useless write-backs can save up to 19.8% of energy and improve endurance by up to 26.2%.

I. Introduction

The emergence of cloud computing, together with the rapidly increasing use of mobile devices that access web-based information, has caused a noticeable growth in the number and size of data centers that house the server infrastructure required by the cloud. With an estimated annual cost of 7.4 billion dollars for 2011, energy consumption has become a primary factor in the design of computer systems targeted at data centers [1]. The memory subsystem accounts for 20% to 40% of the energy consumed by a typical server, making memory an important target for improving the energy efficiency of data centers [2], [3], [4], [5], [6].

In these computer systems, the best choice for main memory in the past three decades has been DRAM due to its low cost per bit and low energy consumption. Previously, static power consumption in DRAM was low enough that it did not influence the total energy consumed. However, with the advent of the multi-core era, where dozens of applications share the same memory, the memory capacity of servers has been extended to the point that DRAM static power accounts for a significant portion of energy. In addition, today's small transistor sizes leak comparatively more current, which leads to even larger static power consumption.

To address these issues, Phase Change Memory (PCM) has been proposed as a replacement for large main memory systems due to several characteristics. Since PCM is non-volatile, power to memory banks can be cut off without fear of losing stored data. This can reduce the static power consumption to negligible levels. The energy required to read data from PCM is also lower than DRAM, while read performance is expected soon to be comparable to DRAM. In addition to energy advantages, PCM will be more scalable than DRAM [7].

PCM, however, does have disadvantages. PCM cells endure only a limited number of writes, typically between 10^7 and 10^8, before failing. Although PCM is more durable than other wearable memories, it still does not have enough endurance to be used in main memory without a wear-leveling or write-filtering mechanism [8], [9], [10], [11], [12]. The power and energy consumption of PCM writes is another major issue that provides several opportunities for improvement.

In this paper, we introduce the concept of *useless write-backs* to analyze the potential that avoiding unnecessary writes can have on energy and endurance. We define a write-back to a lower level of a cache hierarchy to be *useless* when the data that is written back is not used again by the program (i.e., it is *dead*). Since the data is not needed again, the write-back can be safely elided.

Avoiding useless write-backs is possible so long as the information about dead memory regions is available. The technique to obtain and use such information depends on the type of memory region where the data is located. For example, when a call to *free()* is made, a dead region can be identified in the heap. Dead regions in global data can be determined through control flow analysis performed by the compiler, the run-time environment or programmer annotations/hints. Before developing these techniques, it is important to understand the potential that useless write-backs have on energy and endurance. Consequently, the focus of this paper is to determine the maximum amount of useless write-backs for three types of memory regions and their benefit to reducing write energy and improving endurance for PCM.

This paper makes the following contributions:

- We introduce the concept of useless write-backs to drive the development of techniques that can help save energy and improve endurance in PCM main memory.
- We describe an analysis framework to determine the potential impact that techniques based on useless write-backs can have on energy consumption and endurance.
- We use the analysis framework on a set of programs and show that useless write-backs have the potential to save up to 19.8% of energy and improve endurance by up to 26.2%.

The rest of this paper is organized as follows. Section II gives background information on PCM and motivates this work. Section IV explains the analysis framework used to study useless write-backs in programs. Section V presents the results we obtained with the framework. Section VI discusses how the findings of this paper can be applied to a real system. Section VII discusses related work and Section VIII concludes.

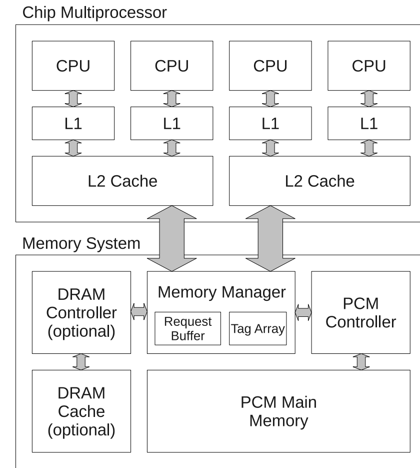

Fig. 1. Hybrid Main Memory Architecture

II. Background and Motivation

A. Hybrid Main Memory Architecture

Previous proposals that use PCM in main memory have mostly assumed a hybrid memory architecture, where a small DRAM cache is backed by a larger capacity PCM memory [13], [10], [11]. Figure 1 shows such an architecture, which we assume throughout this paper. The memory operations issued by the CPU are serviced by the L1 and L2 caches. Misses to the L2 cache, as well as write-backs from the L2 cache, are sent off-chip to the memory subsystem. A hybrid memory works as a traditional write-allocate cache (DRAM) with write-back (to PCM) and LRU replacement. Note that when a dirty block is evicted from the DRAM cache, it must be written back to the PCM.

The use of a small DRAM cache to filter accesses to the PCM has two main benefits. First, by coalescing a sequence of writes for the same block into writes to the DRAM cache, this scheme can mitigate part of the negative impact that PCM writes have on energy consumption and endurance. Second, a small DRAM cache will not harm the overall energy consumption.

PCM has also been proposed as a full replacement for DRAM in main memory, i.e., without a DRAM cache between the processor and PCM [8]. Although this architecture cannot benefit from the filtering effect of a DRAM cache, it consumes less overall energy (no DRAM energy consumed) and could be used in systems that are willing to trade off performance in favor of reduced energy consumption. For this type of architecture, reducing the number of writes is even more important, not only because it targets low-power systems but also because a way is needed to offset the negative effects brought by the lack of a cache.

B. PCM Writes

PCM writes modify the physical state of a phase-change material between amorphous and crystalline. A logical value of zero (one) is stored by changing the phase of the material to the crystalline (amorphous) state. PCM reads are straightforward as they involve only identifying the state of the phase-change material, which can be done at DRAM speed. The energy for a PCM read is even smaller than for a DRAM read. On the other hand, PCM writes are much more energy intensive than DRAM writes and take more time, due to the need to change the state. Furthermore, changing the physical state of the phase-change material repeatedly has the unfortunate effect of wearing the storage cell until it eventually renders the cell inoperable.

Given the unfortunate effects of PCM writes, avoiding a fraction of them saves energy and improves endurance. Although researchers have suggested several ideas for dealing with PCM's endurance limitations, the question of whether they work as expected in commercial memories is still open. Using simulations, several studies have presented techniques that guarantee memory will not wear out before 7 years, which is more than the expected lifetime of DRAM [9], [10], [11], [12]. However, real devices could still be limited to shorter lifetimes, perhaps due to unforeseen effects in the fabrication of large scale memories or memory access patterns not covered by the simulations. Until this question is answered, it is worthwhile to continue looking for alternatives that extend the lifetime of PCM.

III. Useless Write-Backs

A *useless write-back* is a write-back of a dirty cache block that belongs to a dead memory region. If a cache block that is part of a dead region is dirty in the cache, it will eventually be evicted and written back to memory. However, since we know that the data is not used again (because it belongs to a dead memory region), we can safely avoid doing this write-back.

Figure 2 shows a series of accesses to memory and how they affect the state of a direct-mapped cache (with write-back) that can store 8 words of memory. First, memory location A is written, which causes the block to be allocated in the cache and marked dirty. Then, the last value written to A is read. Third, memory location B, which maps to the same cache block as A, is read. This causes A's data to be evicted and

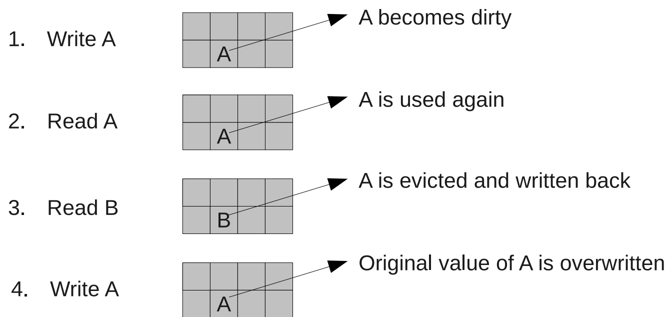

Fig. 2. Example of a useless write-back

written back to the next lower level of the hierarchy. Fourth, memory location A is written again, which as seen by the program, means that the original value stored at location A is overwritten. At this point, we know that the previous value stored at memory location A was not needed since it was last read in the second step. That is, memory location A is dead between steps 2 and 4. The write-back of A's value caused by reading location B is *useless* because location A gets a new value in step 4.

The concept of useless write-backs is independent of the type of memory region to which a particular cache block belongs. The block could belong to the heap, the stack, the global store or any other kind of memory region. In order for the concept to be useful, however, information about the dead regions of memory must be obtained. In this case, the type of memory region is significant because the techniques to determine whether a region is dead are different for each type of region. For example, one way to mark a memory region belonging to the heap is to follow calls to *free()*. After *free()* is called (and before the freed block or part of it is reused again by a call to *malloc()*), the memory region that was returned to the allocator is dead. Clearly, this scheme cannot be used to determine dead regions belonging to the stack or the global store.

This paper focuses on three types of memory regions, namely: heap, global and stack. For each type of region, we assume that a realistic class of techniques will be used to determine dead memory regions. However, this paper does not actually describe these techniques, but instead measures the potential impact that they could have to point the way for further research. Below, we describe how the concept of useless write-backs can be used in the context of each of the three memory region types.

A. Heap

For the heap, we assume that data becomes dead when a call to *free()* returns a block to the memory allocator. Even though other techniques (e.g., control flow analysis) could be used to mark dead regions in the heap, the extensive use of pointers inherent in the manipulation of heap data often makes this analysis inaccurate or overly conservative [14], [15]. Instead, we used a simpler way to identify dead regions in the heap. This technique should indeed be easy to implement, as finding the correct places in the code to mark the beginning of dead memory regions is already done by the programmer through calls to *free()*.

B. Global

We assume that the identification of dead global memory regions is done through control flow analysis performed by the compiler or the run-time environment. This type of analysis can determine the position in the code where a variable becomes dead. Since the memory location of a global variable does not change during the execution of the program, information about dead variables can be translated into information about dead memory regions.

C. Stack

Function activation records held in the run-time program stack can be used to identify dead regions of memory. When a function returns control to its caller, the activation record that was allocated on the stack (to hold return address, frame pointer and local variables) is deallocated and becomes dead. In addition, during the execution of a function, other temporary objects can be placed on the stack, whose memory can be declared dead when they are no longer used by the function, or when the function returns.

To capture these two kinds of dead regions, we chose to use a simple scheme in which all data that is below the stack pointer (assuming the stack pointer grows downwards) is dead. In this case, there is only one dead region of memory, which is delimited by the current stack pointer and the minimum value that the stack pointer has had during the execution of the program.

IV. Analysis Framework

To study the behavior of programs, we developed a framework that determines the potential energy savings and endurance improvement of avoiding useless write-backs. We expect that avoiding useless write-backs does not have a significant influence on performance because writes can be buffered and are, therefore, not on the critical path of execution. Although a large number of buffered writes could potentially cause some reads to be stalled, we expect this situation to be relatively rare. For this reason, the framework does not measure the impact of avoiding useless write-backs on program performance.

Figure 3 gives an overview of our framework and the process involved in analyzing programs. The instrumentation phase gathers a trace (address and type of each memory reference), as well as other information required by the particular type of memory region being analyzed. Then, the trace is fed to the analyzer, which consists of a cache simulator, analysis routines and data structures that maintain information about dead regions of memory. The analyzer outputs the number of useless write-backs and other data that are used to compute the energy savings and endurance gains.

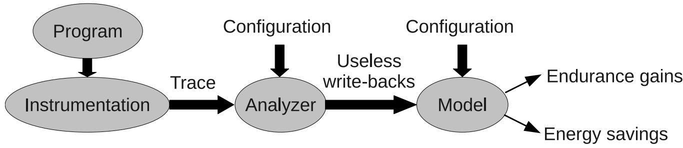

Fig. 3. Framework for measuring gains of avoiding useless write-backs

A. Heap

For the heap, aside from the trace, the instrumentation phase collects the parameters and return values of each call to *malloc()*, *calloc()*, *realloc()* and *free()*. Calls to *malloc()* and *calloc()* are treated the same way, whereas calls to *realloc()* are treated as a call to *free()* followed by a call to *malloc()*.

We keep information about all allocated regions of memory, as well as all regions that have been marked as dead. The cache simulator is used to determine the addresses of evicted blocks that need to be written back. Algorithm 1 counts the number of useless write-backs by performing one of the following actions for each trace entry collected by the instrumentation phase:

- If a *malloc()*, add the start address of the allocated block, together with its end address, to the list of allocated blocks. In addition, if the newly allocated block is part of a previously deallocated block that is now considered a dead region, remove the corresponding part from the list of dead regions.
- If a *free()*, remove the entry from the list of allocated blocks and insert it into the list of dead regions.
- If a memory reference, simulate the access in the cache simulator to get its write-back address, if any. Check whether the write-back address is contained in the list of dead regions. If it is, increment the count of useless write-backs.

B. Global

Our approach for counting useless write-backs for the global region treats each memory word in the global store as an object, and performs an analysis for each object based on the trace, as follows.

Given a trace, a range during which an object is dead starts at the last read before any write and ends at the corresponding write. This is because the object is not read again before it is overwritten. If there is no read between two writes, then the object is dead between both writes. In this case, the first write (and not just the write-back) was unnecessary. Note that many different dead ranges for an object can occur in one trace.

The complication with this scheme arises from the fact that the memory reference trace is read sequentially. Therefore, we cannot know that an object has become dead until we process the last write. By that time, it is possible that the object was evicted from the cache and we could not have incremented the count of useless write-backs because we did not know yet that it was dead. To solve this problem, we use a timestamp to keep track of the last time that a word was evicted from the cache. Once we know the dead range for an object, we can

Algorithm 1 Counting the number of useless write-backs for the heap

```
/*Global data structures*/
/*cache: cache simulator*/
/*allocated: list of allocated memory blocks*/
/*dead: list of dead memory blocks*/
/*useless: number of useless write-backs*/
while hasNextEntry(file) do
    entry ← readNextEntry(file)
    if entry.type = MALLOC then
        allocated.add(entry.start, entry.end)
        if dead.contains(entry.start, entry.end) then
            dead.remove(entry.start, entry.end)
        end if
    else if entry.type = FREE then
        t ← allocated.remove(entry.start)
        dead.add(t.start, t.end)
    else if entry.type = ACCESS then
        writeBack ← cache.access(entry.address)
        if dead.contains(writeBack, writeBack + BLOCK_SIZE) then
            useless ← useless + 1
        end if
    end if
end while
```

check whether the eviction and write-back was indeed useless or not.

Algorithm 2 counts global useless write-backs. As in the heap case, the instrumentation phase records the address and type of every memory reference. The addresses of global objects can be determined by looking at the .data and .bss sections of the binary image. Lists holding the addresses, timestamps of last access and timestamp of last eviction from the cache are kept for all global objects. For each access to global memory, if the last access to that object was performed before the object was last evicted from the cache, we increment the count of useless write-backs. Regardless of whether it was a useless write-back or not, we update the timestamp of last access to that object with the current timestamp. Then, we simulate the cache for the current memory access and, if the write-back address belongs to a global object, we update its last time of eviction with the current timestamp.

C. Stack

For the stack, in addition to the memory trace, we record the value of the stack pointer before each memory reference. As with the heap and global cases, a cache simulator is used to determine the write-back address of evictions from the cache. In addition, we also keep the minimum value of the stack pointer.

Algorithm 3 counts useless write-backs from the stack region. For every entry in the trace generated by the instrumentation phase, we check whether the current stack pointer is less than the minimum recorded value of the stack pointer

Algorithm 2 Counting the number of useless write-backs for global data

```
/*Global data structures*/
/*cache: cache simulator*/
/*globalWords: list of global words*/
/*lastAccess: list of last access to each global word*/
/*lastEviction: list of last eviction of each global word*/
/*icount: current timestamp*/
/*useless: number of useless write-backs*/
while hasNextEntry(file) do
    entry ← readNextEntry(file)
    if globalWords.contains(entry.address) then
        if entry.isWrite then
            if lastAccess(entry.address) < lastEviction(entry.address) then
                useless ← useless + 1
            end if
            lastAccess(entry.address) ← icount
        end if
    end if
    writeBack ← cache.access(entry.address)
    if globalWords.contains(writeBack) then
        lastEviction(entry.address) ← icount
    end if
end while
```

Algorithm 3 Counting the number of useless write-backs for the stack

```
/*Global data structures*/
/*cache: cache simulator*/
/*minStackPtr: minimum value held by the stack pointer*/
/*useless: number of useless write-backs*/
while hasNextEntry(file) do
    entry ← readNextEntry(file)
    if entry.stackPtr < minStackPtr then
        minStackPtr ← entry.stackPtr
    end if
    writeBack ← cache.access(entry.address)
    if minStackPtr ≤ writeBack and writeBack + lineSize < entry.stackPtr then
        useless ← useless + 1
    end if
end while
```

seen so far. If it is, we use this value as the new minimum. Then, we simulate the cache access and record the address of the write-back, if any. Finally, we check whether the write-back address is between the current and the minimum stack pointer. If it is, we increment the count of useless write-backs.

D. Energy and Endurance Models

To evaluate the impact of avoiding useless write-backs, we developed models to derive the energy savings and endurance from data collected by the analyzer. These models are simple, fast to evaluate and accurate enough to measure the potential improvements of avoiding useless write-backs.

Since static power consumption does not directly benefit from useless write-backs, we chose to model only the dynamic component of the energy used by the memory system. We assume that each memory operation consumes a constant amount of energy and the total energy spent is the sum of the energies of all memory operations:

$$E = Cost_{read} \times Reads + Cost_{write} \times Writes$$

Since we are interested only on the savings relative to a baseline, we can treat the actual cost of a memory operation abstractly. However, we do need the relative costs of reads and writes. Assuming that the cost of writes relative to reads is $Cost_w$ and denoting *orig* and *opt* as subscripts for the original and optimized number of reads ($Read$) or writes ($Write$), the energy savings are as follows:

$$E_{savings} = \frac{Reads_{opt} + Cost_w \times Writes_{opt}}{Reads_{orig} + Cost_w \times Writes_{orig}}$$

We use the endurance model of [9], [10] to compute the lifetime of PCM, assuming that a program is run back-to-back for as long as the memory lasts. Given the number of writes that a program makes during one execution, we can calculate how many times the program can be run before the memory expires. Assuming that writes are uniformly distributed across the whole memory (due to a wear-leveling scheme), the lifetime of the memory can be calculated as follows:

$$L = \frac{W_{max} \times T_{exec}}{W}$$

where W_{max} is the total number of times the memory can be written, T_{exec} is the period of one execution of the program and W is the number of writes during one execution of the program. Since $W_{opt} = W_{orig} - W_{orig} \times F_{useless}$, the endurance improvement for a given fraction of useless write-backs ($F_{useless}$) is

$$L_{gain} = \frac{L_{opt}}{L_{orig}} = \frac{W_{orig}}{W_{opt}} = \frac{1}{1 - F_{useless}}$$

V. Experimental Results

A. Methodology

We used our analysis framework on a subset of the SPEC CPU2006 benchmark suite[1] to determine the potential that a technique based on useless write-backs could have on energy and endurance. Our implementation of the instrumentation phase uses Pin [16] to generate the trace of memory references. In addition, Pin includes other information as part of the trace, such as the parameter and return values of calls to memory allocation routines and the value of the stack pointer register before the execution of an instruction. Pin was configured to trace data as well as instruction references.

[1]We could not run *xalancbmk*, *dealII* and *wrf* due to limitations in the program instrumenter and simulator.

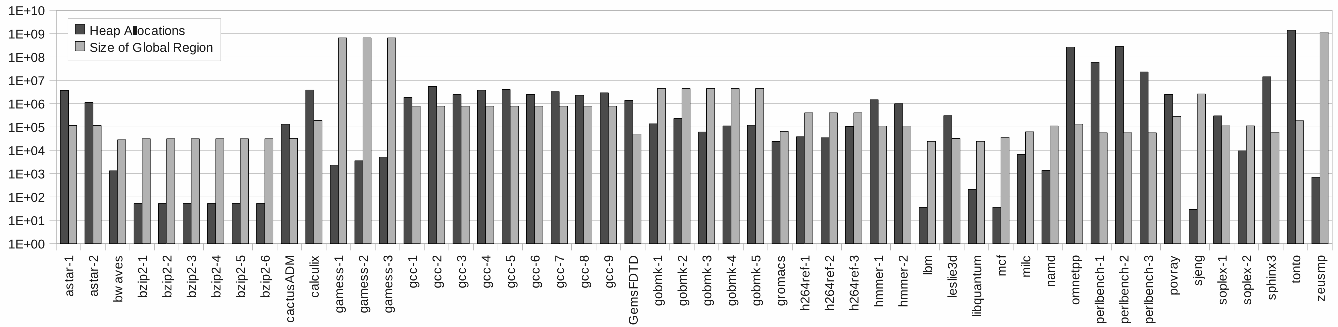

Fig. 4. Number of blocks allocated in the heap and size of the global region in bytes

In the analyzer of the framework (see Figure 3), we used a cache simulator to simulate the L2 and DRAM caches. The L2 is a 1MB, 8-way set associative cache with an LRU replacement policy. The DRAM cache is 16-way set associative, also with LRU replacement. We simulated 5 different configurations, one with no DRAM cache (the processor accesses PCM directly) and 4 others with different DRAM cache sizes (8MB, 16MB, 32MB and 64MB). To explore the effect of the cache line size, we simulated 4 different cache line sizes: 8B (the same size as the processor word), 32B, 64B and 128B. We used a 8B cache line as a baseline for the maximum potential that avoiding useless write-backs can achieve.

The benchmarks were run for 100 billion instructions for the heap and global regions. The results for the stack memory region were obtained by running the benchmarks for 10 billion instructions.

The parameters of the energy model were obtained with an accurate simulator for hybrid memories [11]. The simulator was configured with realistic parameters for future DRAM and PCM memories based on [17], [18] and [19]. We obtained the energy consumption for each type of memory access, which yielded the following results: DRAM reads and writes consume 13.63 and 14.12 times more energy than PCM reads, respectively, while PCM writes consume 36.46 times more energy than PCM reads. Although we do not report the results due to page length limitations, we also tried less aggressive, older energy parameters in our experiments. We found that the results were within 2% of the future parameters and that our conclusions hold for this case as well.

B. Results

Programs exhibit very different characteristics in the way they used each type of memory region. For example, some programs make extensive use of heap data structures, while others prefer to keep all data as global variables. For this reason, the number of useless write-backs for each type of memory region differs from program to program. Since each benchmark benefits more from one particular type of memory region (or none), we categorized benchmarks into three distinct groups: (i) those that benefit knowing about useless write-backs from the heap region, (ii) from the global region, and (iii) those that do not benefit from any region. The reason for not having a category for the stack region will become apparent at the end of this section.

We used the number of allocated objects as a measure of the likelihood that a program will benefit from avoiding useless write-backs from the heap region. For the global region, we used the total amount of space allocated to global variables. Figure 4 shows these two metrics for all 52 test cases, collected from a complete run of each benchmark. Benchmarks with different inputs are treated as different test cases. We classify as *heap-intensive benchmarks* the cases that have more that one million object allocations and the cases with more than 50GB of total space requested to the heap (the latter is not shown in the graph). We classify as *global-intensive benchmarks* those cases with more than 4MB of global object space.

While presenting results for each type of memory region, we give only the results for those benchmarks that are intensive in that particular memory region. In all cases, the improvements for non-intensive programs were negligible and uninteresting. When reporting the average of some value over a subset of the benchmarks, we use the weighted average. For example, the average fraction of useless write-backs is calculated by summing the useless write-backs of all benchmarks and dividing by the sum of write-backs over all benchmarks.

1) Heap: There are a total of 21 (out of 52) test cases from 10 (out of 26) different benchmarks that are classified as heap-intensive. Figure 5 shows the fraction of write-backs from the L2 to the DRAM cache and from the DRAM cache (64 MB) to PCM that are useless. Some benchmarks have a greater fraction of useless write-backs from the L2 cache and others from the DRAM cache. This difference is likely due to allocation sizes and patterns in which data is allocated, written and freed. On average, the fraction of write-backs that is useless is 20.8% for the L2 cache and 8.4% for the DRAM cache.

Figure 6 shows the DRAM and PCM energy savings for the heap-intensive benchmarks. Due to the different data access patterns, the savings vary from case to case, even among different inputs to the same benchmark. On average, the energy savings are 7.8% for the DRAM and 6.6% for the PCM.

An interesting observation about these results is that there

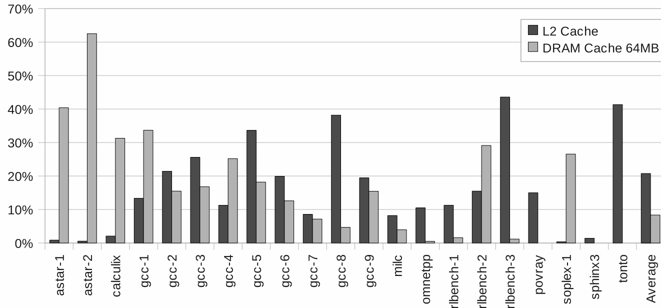

Fig. 5. Fraction of useless write-backs for the heap region (8B cache line size)

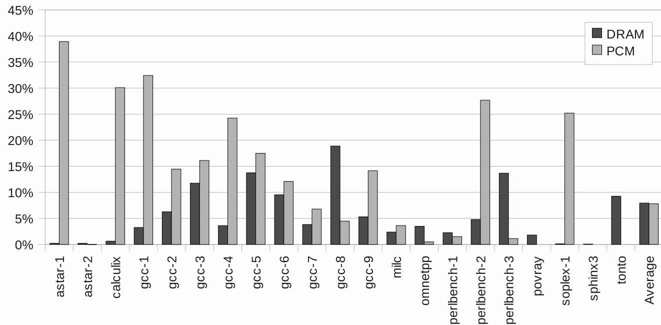

Fig. 6. Energy savings due to useless write-backs for the heap region (64 MB DRAM cache and 8B cache line size)

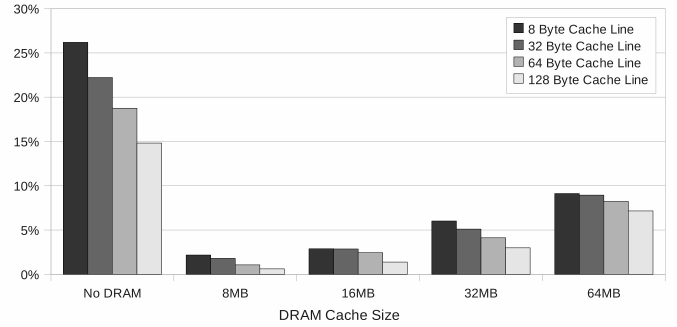

Fig. 7. Average endurance gains from the heap region for different cache line sizes and DRAM cache sizes.

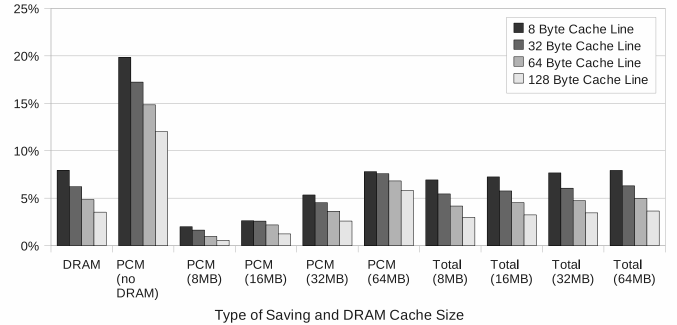

Fig. 8. Average energy savings from the heap region for different cache line sizes and DRAM cache sizes.

appears not to be a good correlation between the fraction of useless write-backs and the energy savings for some of the benchmarks. For example, *astar-2* has a very high fraction of useless write-backs (62.5%) but very low energy savings (less than 0.1%). The reason for this is that the proportion of writes to PCM relative to DRAM for these benchmarks is very small. Therefore, even if we save a large fraction of PCM writes, these are still too few compared to DRAM. Energy consumption in these benchmarks is dominated by DRAM, which means that avoiding PCM writes has a small impact on energy consumption.

The results of the previous two figures do not fully reflect the constraints that realistic hardware can place on the granularity of the dead memory regions because it assumes that each word of memory is the same size as the cache line size. Figures 7 and 8 show the average endurance improvement and energy savings for 5 DRAM cache sizes (including no DRAM cache) and for 4 cache line sizes (for both L2 and DRAM caches). The first column of Figure 8 refers to the energy savings of the DRAM cache, columns 2 through 6 to the PCM energy savings for several sizes of DRAM cache and the last 4 columns refer to the total energy savings for several sizes of DRAM cache.

There are three observations that can be made from these graphs. First, as we increase the size of the DRAM cache, the endurance gains and the amount of energy that can potentially be saved also increases. This occurs because a larger cache can keep dirty data for longer periods of time, increasing the likelihood that they become dead. The biggest gain for both endurance and energy, however, is when there is no DRAM cache at all. The reason is that, for the heap-intensive benchmarks, the fraction of useless write-backs from the L2 cache is bigger than from any of the DRAM cache cases. In addition, the difference between the energy consumed by reads and writes is much bigger for PCM than for DRAM.

Second, energy and endurance gains decrease as the cache line size increases. This is the expected behavior since the opportunities for useless write-backs to arise are reduced when dead regions of memory are forced to share cache blocks with live regions (effectively, as cache line size is increased, dead and live data will share the same cache blocks).

Third, total (DRAM and PCM) and DRAM-only energy savings are almost equal. This is caused by a much higher number of write-backs (and therefore useless write-backs) from the L2 to the DRAM cache than from the DRAM cache to the PCM.

2) Global: Even though there is only a small number (9) of global-intensive benchmarks, there is still potential for improvement. Figures 9 and 10 show the fraction of useless write-backs from the L2 and DRAM cache (8MB), and the energy saved in the DRAM and PCM for these 9 global-intensive benchmarks. On average, 7.5% (8.1%) of write-backs to DRAM (PCM) are useless, which translates into energy savings of 2.3% (7.7%). Note that DRAM energy savings are

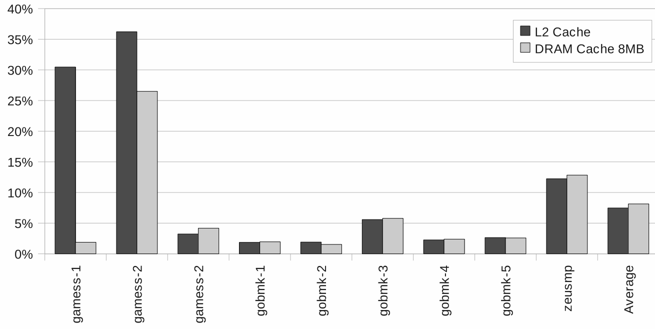

Fig. 9. Fraction of useless write-backs for the global region

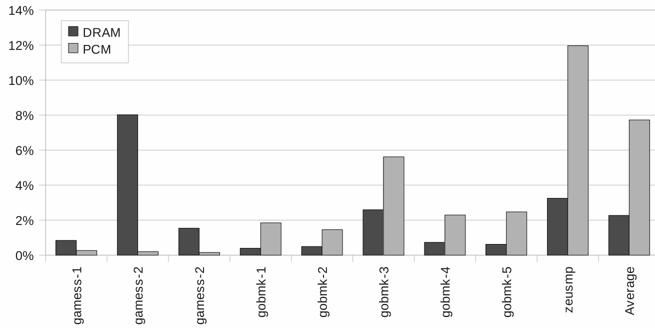

Fig. 10. Energy savings due to useless write-backs for the global region

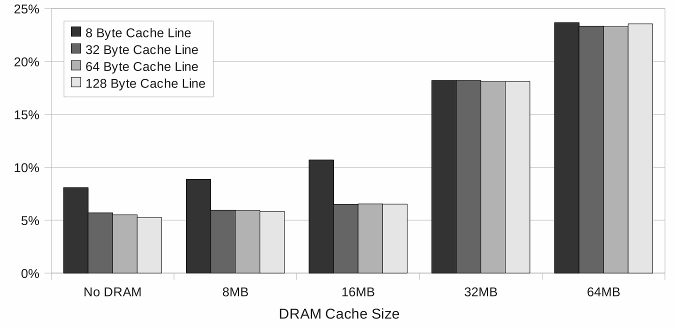

Fig. 11. Average endurance gains from the global region for different cache line sizes and DRAM cache sizes.

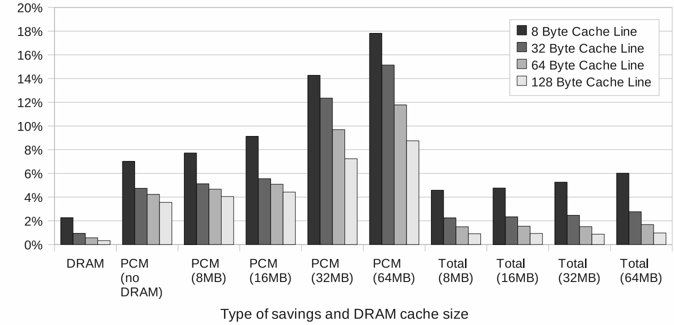

Fig. 12. Average energy savings from the global region for different cache line sizes and DRAM cache sizes.

smaller than PCM, although we save almost the same fraction of write-backs.

Figures 11 and 12 show the average endurance improvement and the average energy savings for global data, similar to Figures 7 and 8. As the cache block size increases, the opportunities of having a dead region that covers a whole cache block decrease, leading to lower energy and endurance gains. However, this effect is not as marked as in the heap region, especially for the 32, 64 and 128 block sizes. This difference is due to the way programs use the heap and global regions. Global-intensive programs tend to allocate big objects (arrays) in the global region, while heap-intensive programs use the heap to store small (scalar and small struct) objects. This makes the heap more sensitive to the size of the cache block.

The total energy consumption is not dominated by the DRAM, as it was for the heap region. The reason for this is that the DRAM cache hit rate is much smaller for the global-intensive benchmarks, which causes an increase in the number of accesses to PCM. In addition, the energy savings for the no-DRAM cache case are smaller than in the heap region. This is due to a smaller fraction of useless write-backs from the L2 compared to the heap region.

3) Stack: The results for the stack region are not as encouraging as those of the other two regions. The maximum fraction of useless write-backs for any of the 52 benchmark-input combinations was only 2.3% for the L2 cache and 0.8% for the DRAM cache. In terms of energy savings, the maximum savings achieved were only 0.9% for the DRAM and 0.3% for the PCM. On average, the endurance gains and energy savings we less than 0.1%.

The reason behind such poor results is that programs generally use only a small part of the stack, with most stack memory references going to the same 10KB or 20KB of memory. Since the stack is so small and can therefore be kept in the cache, there will be very few opportunities for dead data to be evicted from the cache. In addition, dead data in the stack is reused quickly because new functions are called shortly after old ones return. This means newly declared dead regions do not stay dead long enough to be evicted from the cache.

VI. Discussion

Having shown the potential impact of avoiding useless write-backs on energy and endurance, we now discuss how these techniques can be implemented in a real system and what changes to the hardware are needed. We assume that the compiler or run-time environment has the necessary information to determine at what points in the program data becomes dead and that it can insert instructions at these points to convey the address and size of these dead regions to the hardware.

For the case of the heap memory region, this information is available at the memory allocator, which can determine the size of the region being freed. To pass this information to the hardware, the *free()* function would be modified to execute a special instruction with the address and size of the freed

block as parameters. Similarly, for the case of the global region, the compiler or run-time environment can insert calls to this special instruction after it has performed the control flow analysis and has determined the program locations where data becomes dead, as well as their addresses and sizes.

To pass the information about dead memory regions to the hardware, the instruction set could be augmented with a new instruction that enables programs to declare a range of addresses dead. This new instruction would take two arguments: the start address of the dead region and its size. The semantics of this instruction is that the current contents of the memory region referred to by the instruction parameters will not be used again and, consequently, the hardware may discard them. What the hardware actually does with this instruction is implementation defined. In fact, the hardware can choose to ignore the hint and do nothing, in which case the implementation will not benefit from useless write-backs.

If the hardware is to benefit from useless write-backs, it must be able to modify the meta-data stored in the caches. On-chip caches could be modified to provide this functionality, whose implementation could be based on the flush or invalidate operations already implemented on some processors [20], [21]. For off-chip caches, a new memory command could be used to transmit the dead range to the memory manager.

When the new instruction is executed, the hardware should *mark clean* all cache blocks that are part of the new dead memory region and that are currently in the cache, i.e., it should clear dirty bits but keep valid bits set. If any of these cache blocks is later evicted from the cache, it will not be written back because it is now clean. This simple task is enough to avoid useless write-backs to the next lowest level of the cache.

Note that using other common cache operations, such as flush or invalidate, will not have the desired effect. Flushing the cache will actually write back the data, and invalidating it, while still avoiding the write-back, could have a negative impact on performance, because the benefits of temporal locality are lost. This is particularly true the for heap region where memory allocators are actually designed to improve cache locality by reusing recently freed blocks [22], [23], [24], [25], [26].

If the new instruction is to be used in a hybrid main memory, the is an additional consideration that needs to be made. Although it might be tempting to use the scheme only in the DRAM cache without marking blocks clean, it must be noted that this will likely fail to produce any energy savings or endurance gains. The reason is that the dirty block at the highest level can be evicted soon after the hint has been issued, causing the block at the lowest level to become dirty again. For example, consider a dirty block in both the L2 and DRAM caches. If the application declares that block dead but the hardware only marks it clean in the DRAM cache, it will eventually be evicted from the L2 and written back, assuming that the application does not use it for some time. This write-back will cause the block at the DRAM cache to be marked dirty again, canceling out the effect that the declaration of the dead block could have on the PCM memory.

VII. Related Work

Several recent studies have proposed techniques to reduce energy consumption and improve endurance in PCM main memory. These techniques have largely focused on hardware mechanisms. Yang et al. [27] and Zhou et al. [8] use data-comparison write to reduce bit programming. By comparing each bit in the read buffer with the new data bit, the corresponding memory bit is only programmed if the new data bit differs from the old bit. Cho and Lee extend this idea by introducing a bit that indicates whether the associated data is stored negated or not, thereby reducing the number of bit updates [28]. Dhiman et al. [29] and Zhang and Li [13] use wear-aware OS-level page allocation to reduce writes to PCM. Lee et al. proposed multiple dirty bits per cache block to mask off unnecessary memory updates [19]. Qureshi et al. examined a hybrid architecture that uses a DRAM cache to filter accesses to PCM and propose wear-leveling techniques to reduce writes [9], [10]. Ferreira et al. propose page partitioning to reduce the size of the data that needs to be written back [11]. They also propose a clean-preferred page replacement algorithm that gives priority to clean pages when choosing an eviction victim, which reduces write-backs to PCM by coalescing updates in the DRAM cache. In [12], Ferreira et al. propose a swap algorithm for wear-leveling. In contrast to our approach, these techniques do not use application information to improve energy consumption and endurance. These techniques are in fact orthogonal to ours and could be applied to the same system.

Hu et al. propose migrating data to the scratch-pad memory of a different core to avoid write-backs of shared data [30]. This technique uses program analysis to determine when and where to migrate data. Our technique also requires program analysis, although we avoid write-backs by using information about dead memory regions instead of information about how tasks communicate.

Isen and John propose using *malloc()* and *free()* calls to determine intervals where data is dead or uninitialized [31]. They use this information to save energy by avoiding refresh operations and write-backs to DRAM. Our study also uses calls to the memory allocator to avoid write-backs. However, our work focuses mainly on PCM and considers the impact of avoiding write-backs on endurance. In addition, we study other memory regions as possible candidates for improving energy consumption and endurance.

VIII. Conclusion

In this paper, we introduced the concept of useless write-backs and explain how it can be used to improve energy consumption and endurance of PCM main memory systems. We developed an analysis framework to determine the number of useless write-backs for the heap, global and stack memory regions. We also developed a simple energy model to determine the potential energy savings that can be achieved by avoiding useless write-backs. This paper also gives a discussion of how

a scheme based on useless write-backs could be implemented in a real system, with particular attention on the instruction set support and hardware modifications to the memory subsystem.

Using the analysis framework on a subset of the SPEC CPU2006 benchmarks, we determined that a technique based on useless write-backs is a promising candidate for reducing energy consumption and improving endurance of PCM main memory. In particular, the heap and global memory regions seem to offer the most opportunity for improvement. We showed that, on average, useless write-backs have the potential to reduce energy consumption by up to 19.8% and improve endurance by up to 26.2%.

IX. Acknowledgements

This work was supported in part by NSF awards CNS-1012070, CCF-0811295, CCF-0811352, and CNS-0702236.

References

[1] U.S. Environmental Protection Agency, "Report to congress on server and data center energy efficiency - public law 109-431," 2007.

[2] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," in *ACM International Symposium on Computer Architecture (ISCA)*, 2007.

[3] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt, "Understanding and designing new server architectures for emerging warehouse-computing environments," in *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 315–326.

[4] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2009, pp. 267–278.

[5] D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: eliminating server idle power," in *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2009, pp. 205–216.

[6] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi, "Rethinking dram design and organization for energy-constrained multi-cores," in *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2010, pp. 175–186.

[7] "Process integration, devices and structures," in *International Technology Roadmap for Semiconductors*, 2009.

[8] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2009, pp. 14–23.

[9] M. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, 12-16 2009, pp. 14 –23.

[10] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2009, pp. 24–33.

[11] A. P. Ferreira, B. Childers, R. Melhem, D. Moss and, and M. Yousif, "Using pcm in next-generation embedded space applications," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, 12-15 2010, pp. 153 –162.

[12] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Mosse, "Increasing pcm main memory lifetime," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, 8-12 2010, pp. 914 –919.

[13] W. Zhang and T. Li, "Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures," in *PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 101–112.

[14] R. P. Wilson and M. S. Lam, "Efficient context-sensitive pointer analysis for c programs," in *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*. New York, NY, USA: ACM, 1995, pp. 1–12.

[15] B. Steensgaard, "Points-to analysis in almost linear time," in *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1996, pp. 32–41.

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2005, pp. 190–200.

[17] Micron, "Technical note tn-47-04: Calculating memory system power for ddr2," June 2006, http://www.micron.com.

[18] E. Doller, S. Eilert, and A. Camber, "Numonyx seminar talk at Carnegie Mellon University (slides)," September 2009, http://www.pdl.cmu.edu/SDI/2009/092309.html.

[19] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2009, pp. 2–13.

[20] *Intel 64 and IA-32 Architectures Software Developer's Manual*, http://www.intel.com/products/processor/manuals/.

[21] *ARM1136JF-S and ARM1136J-S Technical Reference Manual*, http://infocenter.arm.com/help/index.jsp.

[22] D. Lea, "A memory allocator," http://g.oswego.edu/dl/html/malloc.html.

[23] D. Grunwald, B. Zorn, and R. Henderson, "Improving the cache locality of memory allocation," in *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*. New York, NY, USA: ACM, 1993, pp. 177–186.

[24] Y. Feng and E. D. Berger, "A locality-improving dynamic memory allocator," in *MSP '05: Proceedings of the 2005 workshop on Memory system performance*. New York, NY, USA: ACM, 2005, pp. 68–77.

[25] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos, "Scalable locality-conscious multithreaded memory allocation," in *ISMM '06: Proceedings of the 5th international symposium on Memory management*. New York, NY, USA: ACM, 2006, pp. 84–94.

[26] A. Jula and L. Rauchwerger, "Two memory allocators that use hints to improve locality," in *ISMM '09: Proceedings of the 2009 international symposium on Memory management*. New York, NY, USA: ACM, 2009, pp. 109–118.

[27] B.-D. Yang, J.-E. Lee, J.-S. Kim, J. Cho, S.-Y. Lee, and B.-G. Yu, "A low power phase-change random access memory using a data-comparison write scheme," may. 2007, pp. 3014 –3017.

[28] S. Cho and H. Lee, "Flip-n-write: A simple deterministic technique to improve pram write performance, energy and endurance," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, 12-16 2009, pp. 347 –357.

[29] G. Dhiman, R. Ayoub, and T. Rosing, "Pdram: a hybrid pram and dram main memory system," in *DAC '09: Proceedings of the 46th Annual Design Automation Conference*. New York, NY, USA: ACM, 2009, pp. 664–469.

[30] J. Hu, C. J. Xue, W.-C. Tseng, Y. He, M. Qiu, and E. H.-M. Sha, "Reducing write activities on non-volatile memories in embedded cmps via data migration and recomputation," in *DAC '10: Proceedings of the 47th Design Automation Conference*. New York, NY, USA: ACM, 2010, pp. 350–355.

[31] C. Isen and L. John, "Eskimo:energy savings using semantic knowledge of inconsequential memory occupancy for dram subsystem," in *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. New York, NY, USA: ACM, 2009, pp. 337–346.