Concurrent Migration of Multiple Pages in Software-Managed Hybrid Main Memory

Santiago Bock, Bruce R. Childers, Rami Melhem and Daniel Mossé
Department of Computer Science, University of Pittsburgh
Email: {sab104, childers, melhem, mosse}@cs.pitt.edu

***Abstract*—This paper describes Concurrent Migration of Multiple Pages (CMMP), a new hardware-software mechanism for managing hybrid main memory (DRAM+PCM). CMMP migrates multiple pages concurrently without significantly affecting the memory bandwidth available to applications. CMMP provides a simple interface for the OS to observe memory access patterns. CMMP reduces PCM-to-DRAM transfer bandwidth by copying blocks on-demand. It also reduces DRAM-to-PCM bandwidth by suppressing the transfer of untouched blocks back to PCM. Compared to a state-of-the-art page migration approach for hybrid memory, CMMP improves performance by 14% and reduces energy consumption by 29% on average.**

I. Introduction

Due to relatively low cost and high performance, DRAM has been widely preferred for main memory. With advances in DRAM technology during the past four decades, the size of main memory has increased exponentially. However, DRAM faces serious scalability challenges over the next few years, with high energy consumption and reduced reliability as the primary concerns. Several non-volatile memory technologies, such as Phase Change Memory (PCM), have been proposed as an alternatives for high capacity main memory.

A viable memory architecture for combining DRAM and PCM is *software-managed hybrid memory*, where both DRAM and PCM are addressable by the CPU and managed by the operating system (OS) [1], [2], [3], [4], [5]. Apart from exposing a larger physical address space to applications, software-managed hybrid memory allows greater flexibility to manage memory resources because OS policies can be tailored to the needs of application workloads.

Nevertheless, the performance of software-managed hybrid memory suffers from two main drawbacks, which must be addressed before it can be adopted [6]. First, data is managed at the granularity of pages, which can lead to excessive data movement that steals bandwidth from applications, hurting performance. Second, the OS has low visibility of application access patterns, particularly at the main memory level (below the caches). As a result, the OS cannot react quickly to changes in application behavior, potentially causing poor data migration decisions between DRAM and PCM.

This paper describes a new approach to reduce the bandwidth consumed by data migration and to monitor access patterns with high accuracy and without significant delay. Our approach allows *concurrent migration of multiple memory pages*, which reduces reaction time to changes in application behavior. We also introduce two mechanisms to mitigate bandwidth consumption and interference by migration: *on-demand block migration* and *partial demotion*.

II. Software Managed Hybrid Memory

In software-managed hybrid memory, the OS allocates virtual and physical memory pages and manages memory during workload execution. The OS uses a *page migration policy* to decide whether to migrate pages between DRAM and PCM. Software-managed hybrid memory requires several *mechanisms* to implement page migration, including support for monitoring access patterns and low-overhead migration.

In current comodity systems, applications must be paused during page migration, which can cause an intolerable impact on performance [7]. One optimization allows applications to continue executing during migration. As long as the migrating page remains in the source location, a read can be serviced from that location, and the application can continue executing. However, if the application writes to the migrating page, it must be paused because the copy state of the page is unknown.

To avoid pausing on page migration, Bock *et al.* proposed Concurrent Page Migration (CPM) [7]. This mechanism buffers writes in the last-level cache (LLC) to let applications continue executing even after they write to a migrating page. CPM works by pinning pages in the LLC, which prevents the eviction of dirty blocks that would cause writes to migrating pages in memory. Once a migration has finished, dirty blocks are flushed to the new page.

In CPM, only one page can be under migration at a given time. A single migration can become a bottleneck in multi-core systems, where several independent applications may require migration at the same time. We propose hardware support to overcome this limitation. Our technique enables *multiple simultaneous migrations*, while allowing applications to continue execution during migration, even during writes.

III. Concurrent Migration of Multiple Pages

Concurrent Migration of Multiple Pages (CMMP) is a hardware-software co-designed scheme that provides software-managed hybrid memory the ability to migrate multiple pages at the same time. CMMP allows writes to migrating pages by keeping track of which blocks of a page have already been copied to the destination. On an access to a block, the

978-1-5090-5142-7/16/$31.00

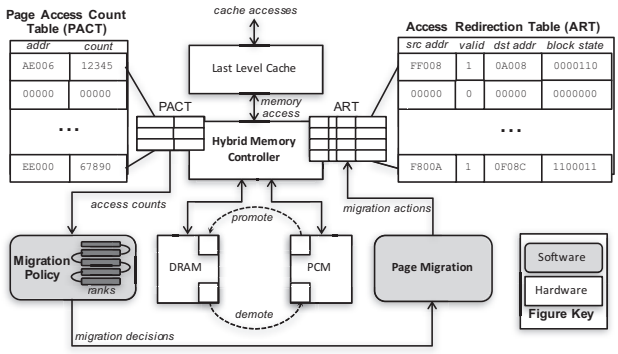

Fig. 1. Overview of software and hardware components for CMMP

state of the block is consulted and the access is redirected to the appropriate location. CMMP inherently supports multiple concurrent migrations by tracking the block states of multiple pages.

CMMP has four parts: Concurrent Migration (CM), Access Redirection (AR), On-demand Block Migration (OBM) and Partial Demotion (PD). CM and AR provide the ability to concurrently migrate multiple pages while allowing applications to continue executing during migration. OBM allows migrations to transfer blocks as they are accessed by the application, minimizing memory system interference. PD allows partially migrated pages to be selected for migration back to PCM, also reducing interference.

Figure 1 shows the design of CMMP. Access counts are collected by hardware, using the Page Access Count Table (PACT). The contents of PACT are periodically read by the OS and passed to the *Migration Policy*. The migration policy, which is implemented in software as part of the OS, ranks pages according to access counts. The policy decides what pages to promote from PCM to DRAM and what pages to demote from DRAM to PCM. Page migration is performed by the Hybrid Memory Controller, which copies pages between memories and redirects memory accesses using the Access Redirection Table (ART) to keep track of the state of migrating pages.

A. Migration Policy

The migration policy is run periodically to generate a list of candidate pages for migration. To make promotion and demotion decisions, the migration policy categorizes memory pages based on access count and recency. CMMP maintains post-LLC access count information using PACT to track accesses to a *subset of pages*. The table is indexed by physical page number; each entry in the table stores how many times the page has been accessed. When accessed, if a page is not found in PACT, a new entry is allocated and the count set to 1. A count of zero is used to identify an invalid entry. When the table becomes full, the entry with the lowest count is evicted. Periodically, the OS reads the contents of PACT and resets the table.

B. Concurrent Migration

CMMP uses ART, a hardware data structure to enable concurrent migration. ART has one entry for each ongoing migration. ART entries are inserted by the OS in response to decisions from the migration policy. After inserting a new entry, the hybrid memory controller copies the page from source to destination, relying on ART to track the state of each block. For PCM to DRAM migrations (promotion), blocks are not transferred immediately to the destination but copied *on-demand* as they are accessed (OBM, see Section III-C). As a result it is possible that some pages are only partially copied to DRAM at a given point of time. For DRAM to PCM migrations (demotion), blocks are copied immediately. OBM is not used for page demotion because pages are cold, and thus, unlikely to have blocks copied as they are accessed.

Next, we describe how promotion and demotion operate, using ART.

1) Promotion: Figure 2 shows the algorithm for PCM to DRAM migration. This algorithm is run for every PCM page that becomes hot during periodic execution of the migration policy, as long as there are free DRAM pages and free ART entries.

2) Demotion: Figure 3 shows the algorithm for DRAM to PCM migration. This algorithm is invoked after the migration policy has determined the next page to demote. DRAM pages that were previously allocated to PCM can reuse their original PCM page to reduce writes. Due to OBM, it is possible that pages are not fully migrated to DRAM when scheduled for demotion back to PCM. CMMP provides support for Partial Demotion (PD), which copies back to PCM only those blocks that have been migrated to DRAM. PD uses the block state information in ART to identify which blocks to copy.

C. On-Demand Block Migration

To reduce migration bandwidth, we introduce On-demand Block Migration (OBM) for CMMP to copy blocks *as they are accessed*. OBM reduces pressure on the memory system in two ways: 1) only blocks that are requested are copied, and 2) bursts of traffic injected by copying the whole page are avoided and spread over more time, effectively giving higher priority to application requests.

OBM uses the block state bitmap in ART to determine the location of the most current copy of a block. On a read access, if the block has not been copied, it is read from the source memory, delivered to the LLC and written to the destination memory. If the block has already been copied, it is read from

```
PROMOTE(pcm_page)
1  if (¬ART.LOOKUP(pcm_page))
2     entry ← ART.INSERT(pcm_page)
3     entry.valid ← 1
4     entry.dest ← dramFreeList.GETPAGE()
5     entry.CLEARSTATEALL()
```

Fig. 2. Algorithm for DRAM to PCM migration (promotion)

```
DEMOTE(dram_page, pcm_page)
 1  entry ← ART.INSERT(dram_page)
 2  entry.valid ← 1
 3  entry.dest ← pcm_page
 4  // Lookup old PCM to DRAM migration
 5  oldEntry ← ART.LOOKUP(pcm_page)
 6  if (oldEntry.valid)
 7     // Old entry exists: copy state and remove old entry
 8     entry.state ← COMPLEMENT(oldEntry.state)
 9     oldEntry.valid ← 0
10  else
11     // No old entry: entire page was copied to DRAM
12     entry.CLEARSTATEALL()
13  for (b ← 0; b ≤ MAXBLOCKS; b ← b + 1)
14     if (entry.GETSTATE(b) == 1)
15        COPY(BLOCKADDR(dram_page, b),
                BLOCKADDR(pcm_page, b))
```

Fig. 3. Algorithm for DRAM to PCM migration

the destination memory and delivered to the LLC. On a write access, the block is written directly to the destination memory regardless of the block state (the latest version is in the LLC).

D. Access Redirection

To allow LLC misses and writebacks to pages under migration to be redirected to their correct location, we incorporate Access Redirection (AR) in CMMP. With AR, migration is transparent and applications can continue executing without pausing, even during writes to migrating pages. AR works after the LLC, inherently supporting multiple cores. Blocks are cached as usual and there are no changes to the caches.

Figure 4 shows the algorithm for AR, which uses ART to determine the current location of blocks. For every LLC miss or writeback, ART is checked to see if the page is currently being migrated. If not, the access is redirected to the corresponding memory based on the physical address of the request (lines 4 to 7). Otherwise, another action is performed (lines 9 to 31) depending on the source page of the request, the type (read vs. write) and the state of the block (copied or not copied).

```
ACCESSREDIRECT(addr, read, data)
 1  page ← PAGENUM(addr)
 2  entry ← ART.LOOKUP(page)
 3  if (¬entry.valid)
 4     if PCM_lowpage ≤ page ≤ PCM_highpage
 5        return pcm.ACCESS(addr, read, data)
 6     else
 7        return dram.ACCESS(addr, read, data)
 8  else
 9     block ← BLOCKNUM(addr)
10     if PCM_lowpage ≤ page ≤ PCM_highpage
11        dram_addr ← ADDRESS(entry.dest, block)
12        if (read)
13           if entry.GETSTATE(block)
14              return dram.ACCESS(dram_addr, TRUE)
15           else
16              entry.SETSTATE(block)
17              d ← pcm.ACCESS(addr, TRUE)
18              return dram.ACCESS(dram_addr, FALSE, d)
19        else
20           entry.SETSTATE(block)
21           return dram.ACCESS(dram_addr, FALSE, data)
22     else
23        pcm_addr ← ADDRESS(entry.dest, block)
24        if (read)
25           if entry.GETSTATE(block)
26              return pcm.ACCESS(pcm_addr, TRUE)
27           else
28              return dram.ACCESS(addr, TRUE)
29        else
30           entry.SETSTATE(block)
31           return pcm.ACCESS(pcm_addr, FALSE, data)
```

Fig. 4. Algorithm for Access Redirection

IV. Experimental Results

A. Methodology

For our experimental evaluation, we use HMMSim, a hybrid main memory simulator [8]. The main simulation parameters are shown in Table I. We model a multi-core system with private L1 caches and a shared L2 cache. DRAM and PCM latencies and energy values are from Lee *et al.* [9]. PACT and ART values were estimated using CACTI 5.3.

For our experiments, we assume a baseline system running CPM [7]. We compare it against three systems that incrementally implement each of the techniques of CMMP: 1) **CM** migrates multiple pages concurrently by copying them as fast as possible. CM uses access redirection to avoid pauses. 2) **CM+OBM** incorporates CM but copies pages on-demand as they are used by the application. 3) **CM+OBM+PD** incorporates CM+OBM and also allows for partial demotion of pages that have not been fully migrated to DRAM.

B. Performance

Figure 5 shows the speedup of each of the techniques of CMMP against the baseline system for all benchmarks. On average, CM alone performs around the same as the baseline, with five workloads having a slowdown of more than 15% (*429.mcf*, *433.milc*, *458.sjeng*, *470.lbm* and *473.astar*). In these workloads, CM migrates too many pages at the same time without controlling the interference caused by migration, which slows down regular application requests. Five workloads have significant speedups with CM: *410.bwaves*, *434.zeusmp*, *436.cactusADM*, *437.leslie3d* and *459.GemsFDTD*. These workloads can increase their share of DRAM requests (from 14% to 26%) with fewer page migrations, reducing interference. For other workloads, additional migrations do not appreciably change the fraction of regular requests serviced from DRAM (52% vs. 49%), resulting in no or low speedups.

When CM and OBM are enabled, performance increases to 10% on average. The bulk of this performance improvement comes from 5 workloads that have more than 25% speedup (*410.bwaves*, *434.zeusmp*, *436.cactusADM* and

TABLE I. SIMULATION PARAMETERS

Parameter	Value
4GHz chip multiprocessor	4 4-issue wide, out-of-order cores, 128-entry reorder buffer
L1 I/D private cache	64KB per core, 4-way, LRU, 3 cycle hit, 16-entry queue
L2 unified shared cache	8MB, 16-way, LRU 32 cycle hit, 32-entry queue
128MB DRAM memory @ 1000MHz	64 banks, 32-entry queue per bank, t_{CAS}-t_{RCD}-t_{RP}: 12-12-12 (ns) Array energy (pJ/bit): 1.17 (reads), 0.39 (writes) Buffer energy (pJ/bit): 0.93 (reads), 1.02 (writes)
4GB PCM memory @ 400MHz	128 banks, 8-entry queue per bank, t_{CAS}-t_{RCD}-t_{RP}: 12-55-150 (ns) Array energy (pJ/bit): 2.47 (reads), 16.82 (writes) Buffer energy (pJ/bit): 0.93 (reads), 1.02 (writes)
PCM/DRAM bus	64-bit single-channel
PACT	8k entries (48KB), 32 set-associative Latency (ns): 0.6 Energy (nJ/access): 0.01 OS read period: 1ms
ART	512 entries (9KB), 32 set-associative Latency (ns): 0.5 Energy (nJ/access): 0.01
SPEC CPU2006 Benchmarks (rate mode)	Working sets larger than 128MB 4 copies run in parallel 1 billion instructions per core 5 billion instructions warm-up phase

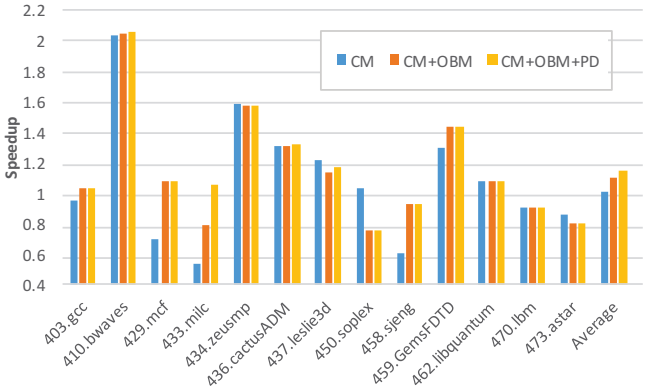

Fig. 5. Speedup of CMMP, normalized to the baseline

459.GemsFDTD). In these benchmarks, the fraction of DRAM requests stays relatively constant. However, the average service time of regular requests to PCM is reduced considerably (from 180ns to 96ns) due to migration of blocks on demand over a large period of time. For workloads with low speedups, the reduction in average PCM access time is not as significant (83ns to 79ns), resulting in similar performance.

Performance is further improved to 14% on average when using PD. Benchmarks with low speedups under OBM benefit little from PD because they suffer mostly from DRAM contention, which PD does not help alleviate. For workloads that do benefit, performance improves because demotions take less time and some writes to PCM can be elided, further reducing contention (average PCM access time is reduced from 70ns to 66ns). In addition, PD releases cold DRAM pages earlier that can be used as destination for hot PCM pages, allowing it to react more quickly to changes in application behavior (fraction of DRAM reads is increased from 18% to 26%).

C. Energy

On average, energy consumption is reduced by 29%. Migration of hot pages to DRAM steers requests away from PCM, which receives 25% less accesses on average. Four benchmarks (*403.gcc*, *450.soplex*, *458.sjeng* and *473.astar*) have a higher energy consumption than the baseline. These workloads perform a very large number of migrations relative to the number of memory accesses, resulting in a overall increase in energy consumption.

V. CONCLUSION

In this paper we propose a novel mechanism for efficient concurrent migration of multiple pages in software-managed hybrid memory. Our mechanism reduces contention at the memory system caused by migration and provides a simple and elegant interface for communicating access counts to software. CMMP reduces migration bandwidth by copying blocks on demand as they are accessed by applications and by eliding the transfer of untouched blocks during migrations to PCM. We evaluate our proposal using multi-programmed benchmarks of different memory access characteristics and show that it can improve performance by 14% on average, while reducing energy consumption by 29%.

VI. ACKNOWLEDGMENTS

The material in this document is based in part upon work supported by the National Science Foundation (NSF) under grant numbers CNS-1012070, CCF-1422331, ACI-1535232 and CNS-1305220. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

[1] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: a hybrid PRAM and DRAM main memory system," in *Design Automation Conference*, 2009.
[2] W. Zhang and T. Li, "Exploring phase change memory and 3D die-stacking for power/thermal friendly, fast and durable memory architectures," in *Int'l. Conf. on Parallel Architectures and Compilation Techniques*, pp. 101–112, 2009.
[3] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories," in *International Symposium on High Performance Computer Architecture*, 2015.
[4] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi, "Operating system support for NVM+DRAM hybrid main memory," in *Workshop on Hot Topics in Operating Systems*, 2009.
[5] L. Ramos, E. Gorvatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Int'l. Conf. on Supercomputing*, 2011.
[6] S. Bock, B. R. Childers, R. Melhem, and D. Mosse, "Characterizing the overhead of software-managed hybrid main memory," in *International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, 2015.
[7] S. Bock, B. R. Childers, R. Melhem, and D. Mosse, "Concurrent page migration for mobile systems with OS-managed hybrid memory," in *International Conference on Computing Frontiers*, 2014.
[8] S. Bock, B. R. Childers, R. Melhem, and D. Mosse, "HMMSim: A simulator for hardware-software co-design of hybrid main memory," in *Non-Volatile Memory Systems and Applications Symposium*, 2015.
[9] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *International Symposium on Computer Architecture*, 2009.