Data Placement in HPC Architectures with Heterogeneous Off-chip Memory

Milan Pavlovic[1,2], Nikola Puzovic[2], Alex Ramirez[1,2]
[1]Barcelona Supercomputing Center, [2]Universitat Politècnica de Catalunya
{milan.pavlovic,nikola.puzovic,alex.ramirez}@bsc.es

***Abstract*—The performance of HPC applications is often bounded by the underlying memory system's performance. The trend of increasing the number of cores on a chip imposes even higher memory bandwidth and capacity requirements. The limitations of traditional memory technologies are pushing research in the direction of hybrid memory systems that, besides DRAM, include one or more modules based on some of the higher-density non-volatile memory technologies, where one of them will provide the required bandwidth, while the other will provide the required capacity for the application. This creates many challenges with data placement and migration policies between the modules of such hybrid memory system. In this paper, we propose an architecture with a hybrid memory design that places two technologically different memory modules in a flat address space. On such system, we evaluate several HPC workloads against different data placement and migration policies, compare their performance by means of execution time and the number of non-volatile memory writes, and consider how it can be applied to the future HPC architectures. Our results show that the hybrid memory system with dynamic page migration and limited DRAM capacity, can achieve performance that is comparable to a hypothetical, hard to implement, DRAM-only system.**

Keywords*—Memory architecture, Memory management, High performance computing

I. Introduction

A growing disparity in the rates of performance improvement between CPU and memory technologies has created a memory wall. So far, its negative impact has been relieved mostly by creating multi-level cache systems. At the same time, the increase of a single thread performance has reached a performance wall due to the inability to increase the operating frequency and to extract the instruction level parallelism. As a solution, the research community and the manufacturers have resorted to the use of multi-core systems in order to increase the performance of the chip.

A recent study has shown that the increase of the number of cores on a single chip puts great stress on the off-chip memory memory system: when executed on a 128-core system, HPC applications require 64 GB of capacity and may require up to 64 GB/s of off-chip memory bandwidth [10]. It is clear that current memory systems will not be able to sustain these requirements when the number of cores begins to increase.

SRAM and DRAM memories could provide the required bandwidth and capacity but at a great price. A large SRAM or DRAM memory would require a high amount of power: technology scaling brings significant increase in leakage power for both SRAM and DRAM and increases the power that is needed to refresh the cells in DRAM. Increasing the bandwidth of the the off-chip DRAM memory would require increasing the signalling frequency of the wires that connect the DRAM to the processor, limiting the length of this connection in order to preserve signal integrity. This would effectively restrict the area on the Printed Circuit Board (PCB) where DRAM chips could be placed, limiting the number of DIMMs that can be connected to a chip. Another alternative to increase bandwidth is to increase the number of channels (or channel width). This leads to the need for more pins on the processor, increasing the size and the cost of the processor itself, and would lead to the increase of the power consumption of the entire memory system. These problems (wire length and pin count) can be resolved by using 3D-stacked DRAM in the same package with the processor. 3D stacking can bring improvements in bandwidth but will offer only a limited amount of DRAM due to thermal dissipation and constrained area [7].

A. Overview of emerging memory technologies

To overcome these limitations, architects are looking into a number of emerging memory technologies that could replace DRAM as the off-chip memory.

Memristor-based memory technologies, like Spin-Transfer Torque Magnetoresistive RAM (STT-MRAM) [2] and Resistive RAM (RRAM) [18] store data as a resistance. They are non-volatile, power efficient and dense compared to standard SRAM and DRAM technologies. Read and write latencies of these memories are still larger than those of SRAM or DRAM, and RRAM cells can sustain a limited number of writes which limits their lifetime.

Another interesting memory technology is Phase Change Memory (PCM) that uses chalcogenide glass and exploits differences between its amorphous (high resistance) and crystalline (low resistance) states to store data. Existing products support only two states, but PCM allows memory cells to have multiple levels of resistance, enabling the storage of more than one bit per cell. The change between the states is achieved by applying high current to the memory cell, and requires more time than with STT-MRAM or DRAM (optimistic estimate claims $\approx$ 100 ns for writing a PCM cell [6]). Compared to other memory technologies, PCM offers excellent density but at a price of limited endurance (small number of writes into a single cell) and high energy that is needed to write the data.

Table I gives a comparison between SRAM and DRAM and the emerging memory technologies. Looking at the characteristics of the emerging technologies, we can see that there is no *silver bullet*: none of the technologies can provide fast access times combined with high density, high endurance and

978-1-4799-2987-0/13/$31.00 ©2013 IEEE

TABLE I. CHARACTERISTICS OF CURRENT AND EMERGING MEMORY TECHNOLOGIES.

Technology	Density	Speed		Endurance
		Read	Write	
SRAM	$60 - 175F^2$	$\approx 0.3ns$	$\approx 0.3ns$	Very High
DRAM	$4 - 15F^2$	$\approx 1ns$	$\approx 0.5ns$	Very High
PCM	$6 - 20F^2$	$\approx 60ns$	$\approx 100ns$	Low
STT-MRAM	$8 - 16F^2$	$\approx 10ns$	$\approx 10ns$	Very High
RRAM	$1 - 4F^2$	$\approx 50ns$	$\approx 250ns$	High

low power consumption. For example, when a PCM is used as a standalone main memory, the system is 1.6x slower and uses 2.2x more energy than a system with DRAM only [6]. To complicate things even more, non-volatile memories either have limited write endurance, or require high current for writing data and system architects should seek to minimize the number of writes to them. An overview of current research trends in hybrid memory systems is given in Section IV.

Out of all emerging memory technologies PCM is the one that is closest to production in large volumes [19], and in the rest of this paper we focus on it. We analyze an architecture with hybrid off-chip memory system that combines small DRAM memory with large PCM memory module. We focus on HPC workloads that have high requirements from the memory system. This paper makes the following contributions:

- Detailed modeling and simulation of a 128-core system with heterogeneous off-chip memory (DRAM and PCM).
- Evaluation of new page migration policies (LRU spill with empty page threshold, lifetime-aware back-migration)
- Analysis of hardware and software static and dynamic strategies for page placement in such a system from the aspect of both performance and the number of writes to the non-volatile memory.
- Analysis of trade-offs between performance and lifetime of non-volatile memories.

II. PROPOSAL

A. System architecture

To stress the memory system as much as possible we focus on a large multi-core chip with 128 processors with L1 and L2 caches and two types of off-chip memory (Figure 1).

Each processor has a private L1 cache while L2 cache is shared and is distributed among cores. The system contains a fast and small DRAM memory, and a large PCM memory. Operating System is responsible for choosing the memory where a new page will be placed, and for making decisions about page migration from one memory to the other. The decisions about page placement and migration may be static or dynamic. In the latter case, OS may require information about pages that is stored in the Memory Management Unit (MMU), such as the number of the accesses to the page or the time the page was last accesses. The details about the page allocation and migration policies are given in the rest of this section.

MMU is responsible for translating virtual to physical addresses, and each processor contains a Translation Lookaside Buffer (TLB) to speed up the translation process. When a decision is reached to migrate the page between memories, MMU will use its DMA engine to perform the data movement. In order to prevent page allocation policies to negatively influence cache effectiveness by altering physical address access patterns, and that way significantly change the total number of requests that reach the main memory, we implemented a simple page coloring mechanism [5].

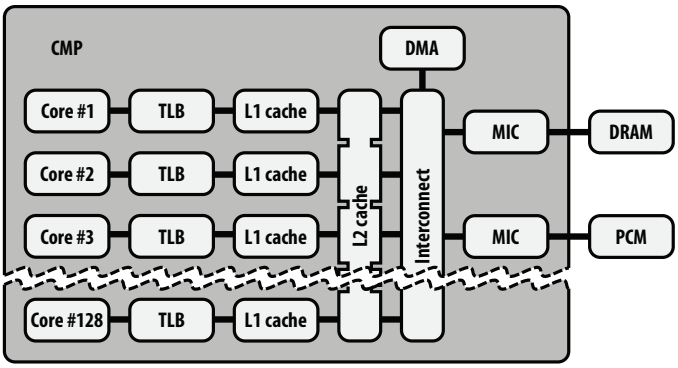


Fig. 1. Target architecture

A migration starts by identifying which TLB holds the translation for a given page. The TLB and caches are then instructed to flush the cache lines that are part of that page. After possible writebacks are completed, DMA starts copying page contents from source to destination address. Finally, TLB is provided with the new translation of the logical page.

B. Static page placement

To better understand the effectiveness of the data placement policies that include migrations, we need to set our baseline using policies that allocate pages without altering their physical location during the application run.

First touch policy allocates pages in order in which they are requested by the cores, first in DRAM, and after exhausting DRAM's capacity, in PCM. The effectiveness of this policy is hugely influenced by the application's access patterns and by the size of DRAM. Performance gains are expected only if the initial access to a hot page happens while there is available space in DRAM. Conversely, if DRAM space becomes polluted with less reused pages, allocated because of their early first access, a performance degradation is imminent [4].

In order to evaluate the full potential of any static allocation policy, and to create the most favorable static distribution of pages between DRAM and PCM, we create a profile of memory accesses for each application under study. A profile is a list of all the logical pages accessed by each core during the execution, with the number of accesses to each page. A page placement policy can then use the profile to "predict" the traffic intensity on a particular page, and decide about its placement. Of course, all the policies that rely on the profile have to pre-run the application, or at least one of its iterations, to generate the profile itself. In many situations, this is impossible or impractical, but the results can still serve as an idealistic baseline in comparison against some other non-profile policy.

Static profile-based policy ensures that the most accessed pages are allocated in DRAM, and least accessed in PCM.

The allocation is done at the beginning of the execution by reading the profile, sorting the pages by their access count in descending order, and allocating them in first in DRAM, and then in PCM. This policy should eliminate some of the downsides of first touch policy. It should bring performance gains by correctly dividing hot and cold pages over fast and slow memory ranges. For the same reason, it should also contribute in decreasing number of writes to PCM, that way extending its lifetime.

To avoid writes to PCM even more, we can modify this policy to take into account only write access count from the profile, instead of the sum of reads and writes. That way, PCM will host pages that are least frequently written regardless of their read traffic. This reduction in number of writes to PCM comes at the expense of performance, as hot read-only pages placed in PCM might bottleneck the system. Nevertheless, depending on the priority between performance and lifetime, this may prove as a justifiable tradeoff.

C. Spill migration

Static allocation policies can not exploit any of the temporal characteristics of the memory access pattern, because the page's initial physical location remains unchanged throughout the execution. The principle of data caching, prefetching and many other migration mechanisms are targeted to alter the distance of a given piece of data from the processor, depending on its potential for temporal reuse.

For this, we propose a spill migration — a policy that performs data movement in one direction only, from DRAM to PCM. Unlike traditional memory hierarchy with cache memories, where data is initially located further from the processor, and then gradually migrated closer as it experiences more reuse, spill migration policy first allocates a page in fast memory (in our case DRAM), and later evicts it to PCM. These evictions are performed due to the limited DRAM capacity, in order to make room for the newly allocated pages. This policy does not account for migrating back those evicted pages that turn to be heavily used after the initial migration has happened — whatever is copied to PCM stays there.

LRU spill policy keeps track of last access time for each page in DRAM, and in case of eviction selects one that is least recently used. The rationale behind this policy lies in the assumption that all the data used by the application throughout its execution time can roughly be divided in two categories: first, data that is reused most of the time, and second, data that is reused in a limited period only, or very rarely reused. If we then assume that DRAM capacity is large enough to fit all the pages that host data from the first category, we can expect that, because of their reuse, they will never be selected for eviction from DRAM. Then, the evicted pages should theoretically be those that are rarely used, and those that completed their high reuse period. The policy expects that the newly allocated pages will experience a significant traffic, at least in the short period after their allocation. Therefore, the gain obtained by initially allocating them in DRAM will justify the cost of migrating them to PCM, if they become less frequently used later.

Eviction from DRAM can take significant time, especially in a system where interconnect and memory can become congested by many simultaneous migrations, each triggered by a memory access from a different core. A naive eviction mechanism would start evicting when one of the TLBs makes a page table miss, and there are no more empty pages in DRAM's address space. It means that the translation of such memory access would have to stall until the eviction is finished, which leads to overall execution performance degradation.

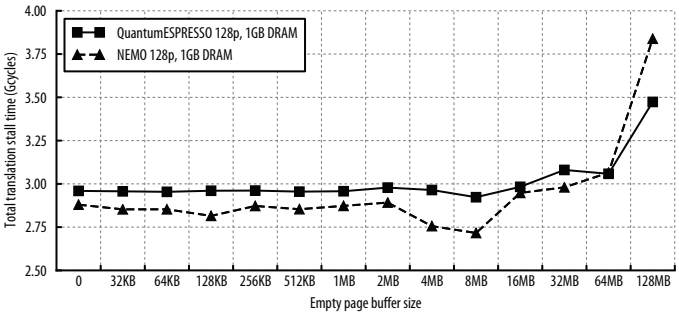


Fig. 2. Empty page buffer size analysis

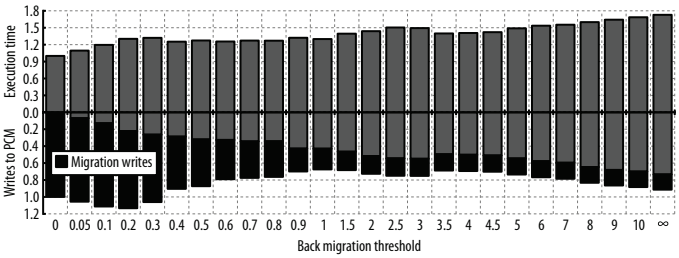
To reduce the number of translation stalls, and to eliminate page eviction time from the translation critical path, we propose triggering the eviction once the number of empty pages in DRAM falls below a selected value — an *empty page threshold*. That way, corresponding TLB can immediately be provided with the translation, and continue processing the request, while the eviction is performed in the background. This can also enable more complex eviction mechanisms, and more detailed last access time analysis, once their operation time is taken off the translation critical path.

Selecting the value of empty page threshold is not trivial. Setting it too low would give TLBs a chance to quickly exhaust a small set of empty pages before any of the scheduled migrations finishes. Setting it too high would effectively reduce DRAM capacity, cause premature evictions, and significant interconnect traffic due to many "on-the-fly" migrations.

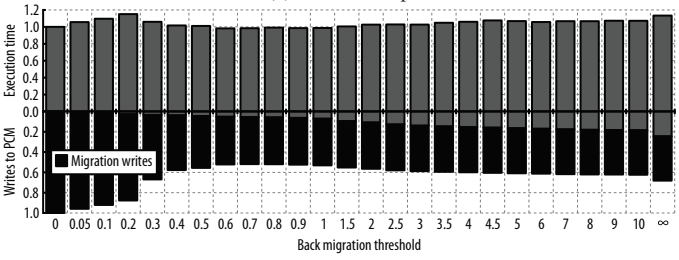
To confirm these claims, on Figure 2 we present overall translation stall time penalty for different empty page buffer size, in a system with 1GB of DRAM, running two different workloads. For low values we notice relatively stable level of total translation stall time. This indicates that the average eviction time is higher than the time in which TLBs exhaust a small supply of empty pages in DRAM. A scenario where one completed eviction unblocks one stalled translation, but shortly after that is followed by another similar translation-eviction pair makes the empty page buffer ineffective, and the changes in its size irrelevant.

On the other side, large empty page buffer allows the TLBs to invoke many more evictions before any of the translations stalls. However, once that happens penalty will be very high, due to a huge number of in-flight migrations and congested interconnect and memory. Measured total translation stall time for large empty page buffers shows that this tradeoff is not beneficial. Moreover, it is unaffordable sacrificing significant DRAM capacity to the empty page buffer, as it can cause other inneficiencies not related to the translation stall time.

Figure 2 shows that the two opposing causes for high overall translation time diminish their influence at around 8MB of empty page buffer size. The sweetspot effect is not overly



(a) NEMO 128p



(b) QuantumESPRESSO 128p

Fig. 3. Back migration threshold analysis

dramatic, but enough to give us a reason to choose this value as fixed for the rest of our experiments.

Similarly to prioritizing writes in static profile-based policy, we also evaluate a modified LRU spill policy, where we select least recently written page for eviction, instead of least recently used. Again, we do that to explore if a decrease in PCM writes can outweigh potential performance degradation.

LRU spill policy suffers from similar drawbacks as the first touch policy — a decision to evict a page from DRAM, and make PCM its final and definitive host, is irreversible, and can be proven costly if suddenly its traffic increases at some later point. To better measure the amount of these wrong evictions and their negative effects, we made use of the profile information to more accurately "predict" expected traffic on a given page. Therefore, **spill profile-based policy** can either spare a page from eviction if its future traffic is high, or victimize it if it is low, regardless of its previous access count. That way, at any moment during the execution, this policy keeps in DRAM those pages that will have most accesses in the future, and use the profile to make ideal eviction decisions and achieve superior performance than LRU spill policy. Aside from being unpractical for use because of the need of a pre-run for generating a profile, this policy is technically hard to implement, because every eviction demands a comparison of each DRAM page statistics against the corresponding entry in the profile. However, as a "perfect spill policy" we can use it to estimate how well LRU spill policy performs.

Once again, for favoring PCM's lifetime instead of overall performance, we evaluate modified spill profile-based policy, where we select a page for eviction based on its future write access count, instead of the total access count.

D. Dynamic page migration

Finally, a **Dynamic policy** introduces page migration in the other direction (from PCM to DRAM), denoted as back-migration. It is an extension of spill policy, so the same rules for eviction from DRAM still stand — whenever a page fault occurs, the system tries to allocate the page in the DRAM. In case an eviction is needed from DRAM, a page is selected in the same way as with LRU spill or spill profile-based policy.

The decision to back-migrate the page is made when the page is accessed in the PCM. When a page is first brought to the PCM we reset its access counter, regardless of how many times it was accessed in the DRAM. At the same time we keep track of the number of accesses for every page in the DRAM, as well as the average for all the pages ($n_{DRAMavg}$). When a page in PCM is accessed, we compare its access counter ($n_{accesses}$) with the average number of accesses to pages in DRAM. If (1) is satisfied, we migrate the page back to DRAM:

$$n_{accesses} > back\_migration\_threshold \times n_{DRAMavg} \quad (1)$$

Back migration threshold (BMT) is a value that controls the aggressiveness of migration triggering. If it is set to zero, a page is migrated as soon as it is touched in PCM, so the DRAM acts as a typical cache. In this case we expect good performance as the system tends to always move active pages to DRAM, but due to a large number of migrations, number of writes to PCM may go high. On the other hand, if BMT is set to infinity the page never gets migrated back, and then the policy is equivalent to LRU spill. In between those extremes we would like to search for values that give good performace and low number of PCM writes.

Figure 3 shows impact of changing BMT from 0 to infinity, on the performance and on the number of PCM writes, when executing two of our applications. Top part of the graph shows execution time normalized to an aggresive migration setup, when BMT is set to zero. Bottom part presents the number of writes to PCM normalized to the same setup. Since the aggresiveness of migrations directly influences number of writes to both memory modules, we separated PCM writes that are part of a migration, from those that are requested from the cores.

We can observe that in both cases the execution time is lowest when BMT is set to zero. This is expected, since in this case DRAM behaves like a cache, and the number of application accesses to PCM is close to zero. Same conclusions have been shown in similar architectures, when migrating pages to on-chip memories [17]. However, due to migrations from DRAM to PCM, the number of writes to PCM is high. As we increase BMT, two applications show different behaviour. However, we can spot a value for BMT of 1 as a rough minimum for the number of PCM writes, which also has a decent performance. In case of NEMO, performance degrades $\sim$30%, but PCM writes decrease for the same value. In case of QuantumESPRESSO, performance stays on roughly the same level, while PCM writes decrease for almost 50%.

This gives us enough reason to further investigate two dynamic migration policies: first, performance-oriented with BMT set to 0, and second, lifetime-oriented, with BMT set to 1. Similarly as with previous policies, we will also investigate the modification that takes into account only write accesses. That is, back-migration is considered only on a PCM write, and performed if the write count is greater than the average number of writes to the pages in DRAM, multiplied by BMT.

TABLE II. OVERVIEW OF DATA PLACEMENT POLICIES

Data placement policy	Profile based	Prioritize writes	Migrations		
			DRAM→PCM	PCM→DRAM	BMT
First touch	No	No	No	No	-
Static profile	Yes	No	No	No	-
Static profile(w)	Yes	Yes	No	No	-
Spill LRU	No	No	Yes	No	-
Spill LRU(w)	No	Yes	Yes	No	-
Spill profile	Yes	No	Yes	No	-
Spill profile(w)	Yes	Yes	Yes	No	-
Dyn perform	No	No	Yes	Yes	0.0
Dyn perform(w)	No	Yes	Yes	Yes	0.0
Dyn lifetime	No	No	Yes	Yes	1.0
Dyn lifetime(w)	No	Yes	Yes	Yes	1.0

TABLE III. OVERVIEW OF SIMULATED APPLICATIONS, THEIR TOTAL MEMORY FOOTPRINT AND TIME OF THE SIMULATED PART.

Application	Domain	Footprint (GB)	Time
NEMO	Ocean modeling	6.54	$\approx$ 2s
CPMD	Computational chemistry	7.48	$\approx$ 20s
PEPC	Plasma physics	8.79	$\approx$ 30s
QuantumESPRESSO	Particle physics	19.75	$\approx$ 40s
GADGET	Astronomy and cosmology	57.43	$\approx$ 1m

Table II summarizes all previously described data placement policies, and presents an overview of their respective key features — usage of the profile information, prioritization of writes, allowed migration directions between DRAM and PCM, and value of BMT (in case back migration is allowed).

III. EVALUATION

Our investigation focuses on the changes in the execution time and in the number of writes to PCM as we gradually increase the size of DRAM, from 512MB up to the size that is larger than the total footprint of the application. We also try to investigate and explain performance differences as we configure memory management unit to allocate pages based on first-touch policy, profile-based policy and on dynamic policies. To evaluate the system, we use *TaskSim*, a trace-driven cycle-accurate CMP simulator validated against Cell BE [13].

The applications that we use for evaluation are listed in Table III. They are production-level HPC codes that use MPI programming model, and we executed them with realistic input sets. These are representatives of scientific applications that are used in today's supercomputers [15].

To obtain the traces we implemented *MemTraceMPI*, a Valgrind [9] tool for tracing load and store instructions of an MPI application. It instruments all the executed instructions, outputting only information about the memory accesses: access type (load or store) and access size in bytes. This information allows for a detailed simulation of memory accesses. In order to simulate non-memory instructions, the tool records the number of instructions that are executed between two consecutive memory accesses. To preserve the dynamic nature of parallel application, *MemTraceMPI* detects calls to MPI functions and records their parameters. This allows us to simulate the communication and synchronization among MPI processes. Since the simulated system is a chip-multiprocessor, communication is performed using memory copies, and synchronization is achieved using standard SMP synchronization primitives.

TABLE IV. OVERVIEW OF SIMULATED ARCHITECTURE PARAMETERS

L1	Capacity:	8KB	**L2**	Capacity:	1MB per core
	Word size:	8B		Word size:	32B
	Associativity:	4-way		Associativity:	32-way
	Latency:	2 cycles		Latency:	20 cycles
DRAM	CL-tRCD-tRP:	8-8-8	**PCM**	4x slower than DRAM	
TLB	Entries:	128	**MMU**	Page size:	8K

The platform used for application tracing was MareNostrum, a supercomputer in Barcelona Supercomputing Center. It is a cluster of JS21 blades, each with 4 IBM Power PC 970MP 2.3 GHz processors. The 4 processors on a node shared 8 GB of RAM, and were connected to a high-speed Myrinet type M3S-PCIXD-2-I port, and two GigaBit Ethernet ports [16].

Large execution times of our applications led to hundreds of gigabytes of traces per application. To keep the trace files at a manageable size, and to maintain acceptable simulation time, we applied trace filtering against a simple configurable cache (512KB direct cache), included in *MemTraceMPI* tool. Only cache misses are written to the trace. Given that our simulated caches are going to have a larger size than the trace filter caches, we are certain that at least the same number of memory accesses will reach the main memory, as with unfiltered traces. This method has been previously applied in the work of Rico et al. [14] resulting with trace size reductions of up to 98%.

Even after filtering, simulation times were unacceptable (in the order of days). To further reduce the size of the traces, we exploited the fact that all applications iterate multiple times over the input set and included only a few iterations [1], [3].

A. Architecture parameters

Simulated architecture closely modeled architecture presented in Section II-A, and Figure 1, with the most important parameters presented in the Table IV.

B. Results

Figure 4 compares execution times on a system with PCM memory only (t_{PCM}) and on a system with DRAM memory only (t_{DRAM}), both of which are of sufficient capacity to fit the entire application's working set. Vertical axis represents execution time normalized to t_{PCM}, and applications under study are aligned along the horizontal axis. We notice that only NEMO and CPMD experience changes in execution time that is equivalent to the performance difference between PCM and DRAM (4x). The rest of the applications put more stress on the interconnect than on the memory system. In those cases, the positive influence of the fast memory is diminished, as particularly is the case with PEPC, and, naturally, we cannot expect that a system with hybrid memory, regardless of the data placement policy, will give a performance improvement.

Figure 5 shows the main results of our simulations. Each subfigure presents one application, testing the effectiveness of our page placement policies, aligned on the horizontal axis.

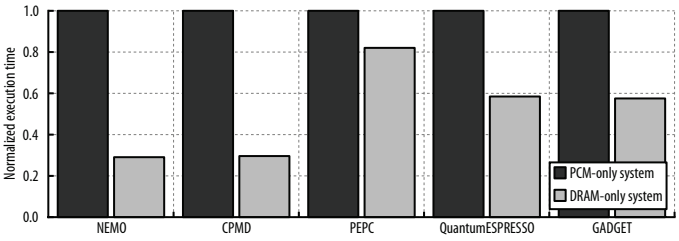


Fig. 4. Performance comparison between PCM-only and DRAM-only system

Top part of each subfigure shows *relative slowdown*, execution time normalized on a range from t_{DRAM} to t_{PCM} using (2).

$$relative\_slowdown = \frac{t_{exec} - t_{DRAM}}{t_{PCM} - t_{DRAM}} \quad (2)$$

Therefore, a relative slowdown with a value of 0 would represent a ”perfect” data placement policy, that yields performance equal to the system with DRAM memory only. Conversely, a system with PCM memory only would give a relative slowdown of 1. On a hybrid memory system that we evaluate, using one of the proposed data placement policies we expect relative slowdown to be between 0 and 1.

Bottom part of each subfigure represents the number of writes to PCM, with a special segment showing writes caused by migrations, normalized to the total writes. Different bars enclosed by each data placement policy segment, depict changes in the aforementioned metrics as the size of DRAM in a system varies from minimal 512 MB, incrementing by a factor of 2 up to a size of the entire application footprint, where any placement policy would be obsolete, because all the data could fit in DRAM. On both vertical axes we imposed a cut-off at a value of 1, so that the low values remain clear. On any bar that is shortened because of this, we write its real value.

As expected, *First touch* policy, with its naive approach, experiences bad performance for all applications unless the size of the DRAM gets very large. Similar results stand for the number of PCM writes, with the only exception occurring when compared to the aggressive migration policies, that can direct huge write traffic to PCM. The other two static policies perform better than the first touch, as they can exploit profile information for improving both of our relevant metrics. It is worth noticing that profile-based policy can reduce slowdown up to 40% using only the smallest DRAM size. Static profile policy that prioritizes writes (*Static profile(w)*) makes a very small, on chart almost invisible, tradeoff against regular static profile policy (*Static profile*), and a slight decrease in number of PCM writes pays with a small performance degradation.

LRU spill policy uses its migration capabilities to outperform first touch policy, but in most cases it is not better than static profile-based. Migrations that it executes do not provoke a significant number of extra writes to PCM, which is good, but also indicates that PCM write traffic is dominated by those pages that are previously evicted from DRAM, which is bad. Confirmation that selecting least recently used page is often a wrong decision comes when comparing it to spill profile policy, which performs in average 20% better, with less PCM writes. Similar to the static policies, spill profile-based policies show little difference if they prioritize writes.

Aggressive migration policy (*Dyn perform*) seems to dominate performance aspect of most applications, even for small DRAM sizes. However, its performance boosting capabilities can severely damage the lifetime of PCM, as the number of PCM writes can grow more than 5 times the baseline, in case of PEPC, QuantumESPRESSO and GADGET. We notice that almost 100% of the PCM writes, regardless of DRAM size, are a product of migrations. This time, modified aggressive dynamic policy (*Dyn perform(w)*) shows more obvious difference, mostly by significantly relieving PCM write traffic, without dramatically reducing performance.

Lifetime-oriented dynamic policy (*Dyn lifetime*) shows a nice balance between performance and number of writes, especially for the DRAM size of 1 GB. It is never too dominant in any aspect, but always among the best policies from what we evaluated. Performance-wise it shows an improvement of 20–60% over PCM-only system, and regarding PCM writes 40–60% (except QuantumESPRESSO and GADGET). It should be noted that lifetime-oriented policy has much more stable and predictable number of PCM writes than aggressive dynamic policy. When directly compared it can reduce PCM writes 20% to 10x, and only marginally degrade performance. Therefore, it might serve well in the wider spectrum of environments.

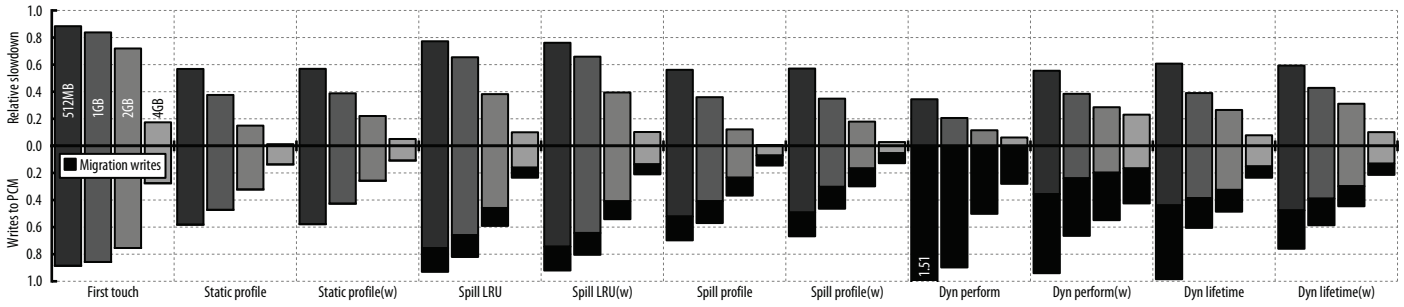
Dynamic policies, however, are not able to easily extract performance benefits with GADGET, the application with the largest footprint, unless DRAM is larger than 4 GB. When compared to any of the spill policies, it becomes clear that the dynamic policies cannot significantly reduce the amount of non-migration PCM writes. A closer insight in access profile information reveals that many pages in GADGET's working set experience similar traffic, and, therefore, cannot be easily divided into ”hot” and ”cold” segments, suitable for placing in DRAM or PCM, respectively. If DRAM is too small, this creates a ping-pong effect, where many pages are repeatedly migrated back and forth between DRAM and PCM.

IV. Related work

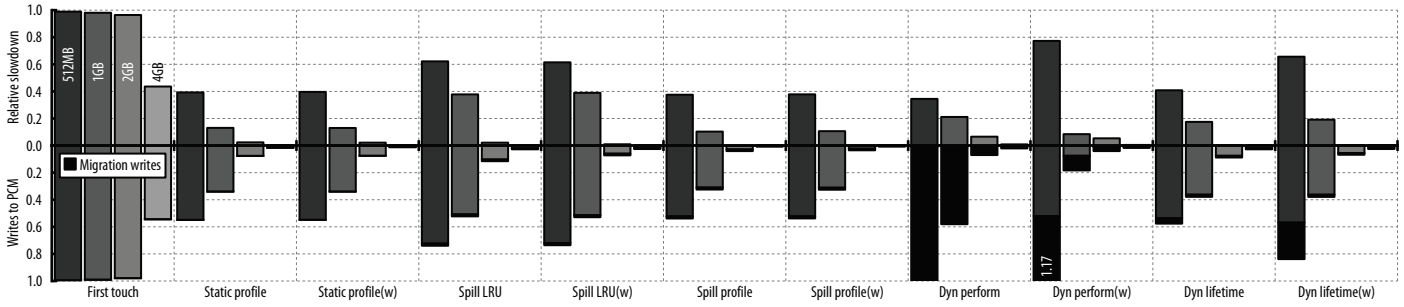
Ramos et al. [12] proposed placing DRAM and PCM in a flat address space, with MC deciding about migrations between them. For facilitating this, they introduced a novel page ranking and migration policy. However, their target architecture did not include more than 8 cores, and they did not evaluate HPC applications for stressing memory bandwidth and capacity.

Qureshi et al. [11] placed DRAM in front of PCM as a cache with the intent to bridge the latency gap. Their work tackled PCM endurance issues by managing lazy-write organization, line-level writes, and wear-leveling. They show that a performance improvement of 3X is achievable with a DRAM buffer of only 3% size of PCM. By placing our memory modules in a flat address space instead, we try to increase the overall memory capacity, avoid negative impact of the DRAM cache on the workloads with low locality, and give the option of further upgrading the memory system with a module of different characteristics than DRAM or PCM.

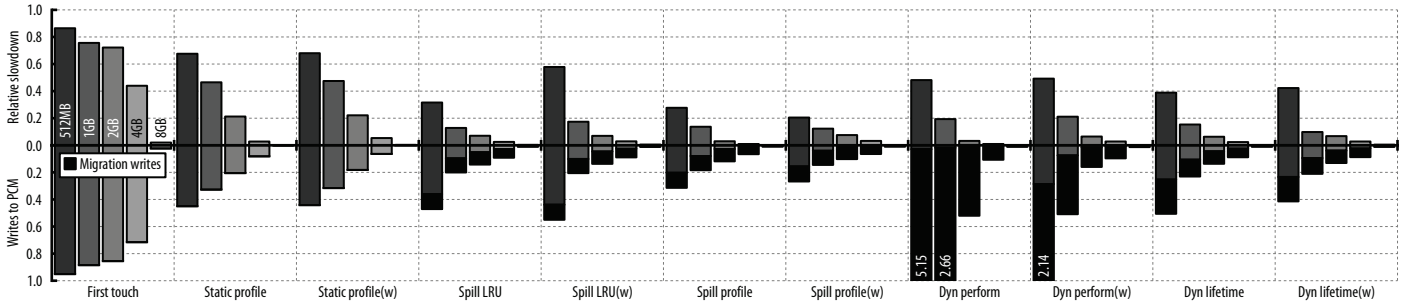
Lee et al. [6] explored another hybrid memory organization with PCM and DRAM as a buffer, and concluded that PCM's long latencies, high energy, and finite endurance can be effectively mitigated. These effective buffer organizations and partial writes make PCM competitive with DRAM at current



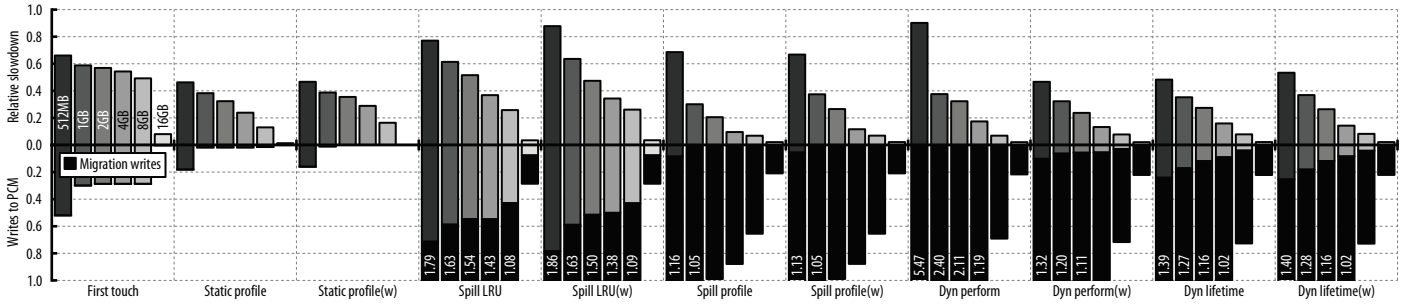
(a) NEMO (128 cores, footprint: 6.54 GB)



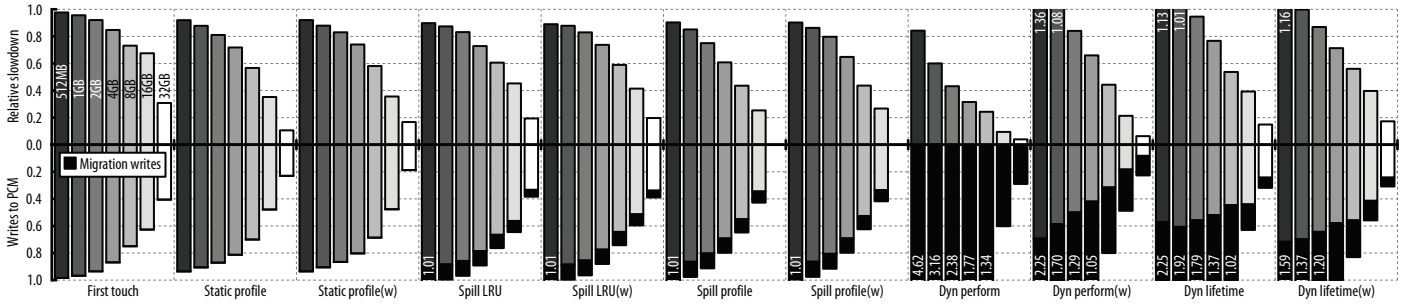
(b) CPMD (128 cores, footprint: 7.48 GB)



(c) PEPC (64 cores, footprint: 8.79 GB)



(d) QuantumESPRESSO (128 cores, footprint: 19.75 GB)



(e) GADGET (128 cores, footprint: 57.43 GB)

Fig. 5. Overall performance and amount of PCM writes comparison of different data placement policies

technology nodes. Moreover, proposed solutions are area neutral, which is a critical constraint in memory manufacturing.

A position paper from Hewlett-Packard [8] argues for the OS support in hybrid memory systems with the combination of DRAM and flash (or PCM) memory. The paper focuses on server workloads, but does not provide a detailed evaluation.

V. Conclusions

There is no *silver bullet* memory technology that allows high bandwidth, high density and low latency at the same time. Hence, many research efforts have focused on designing hybrid memory systems with different types of memory modules that in combination offer the characteristics that are needed. In this work, we looked at one such system that consists of a small and fast DRAM memory, and a large and slower PCM memory. We have focused on the problem of page placement and page migration in such system, and, to the best of our knowledge, we have performed first detailed analysis of several algorithms for performing this task. Using a set of High Performance Computing applications we have analyzed how the design parameters affect both performance of the system and the lifetime of the PCM memory.

Besides analyzing existing page placement algorithms, such as first touch or profile-based policies, we have also developed dynamic algorithms for placement and migration based on LRU spilling and back-migration using either aggressive (for performance) or more conservative (for PCM lifetime) policy.

Our analysis has started off with an expected result: if the only aim is to optimize for performance than the page placement and migration policy needs to be as aggressive as possible and needs to utilize DRAM memory as much as the capacity allows. Our results show that a system with only 1GB of DRAM and an aggressive dynamic policy is only 20–60% slower than the system that has maximum DRAM capacity.

However, the performance of the aggressive dynamic policy comes at a price in the number of writes to PCM. These writes are costly, and they reduce the lifetime of the memory, so we have looked at other policies that aim to reduce their number, while keeping performance at an acceptable level. We have shown that these policies (called *Dynamic lifetime* in our analysis) when compared to the aggressive dynamic policy can reduce the number of PCM writes from 20% to 10x, while reducing performance up to 10%.

In conclusion, we have shown that page placement is a very important aspect of a system with hybrid memory, and the choice of the policy should be taken seriously and should depend on the objectives that the system architect has set.

Acknowledgments

The research that led to this work is supported by the Spanish Ministry of Science and Technology under CICYT project with reference number TIN2012-34557 (Computación de Altas Prestaciones VI), by Mont-Blanc project (European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 288777), and the grant SEV-2011-0067 of Severo Ochoa Program awarded by the Spanish Government. The authors thank the reviewers for their guidance and comments to improve quality of this paper.

References

[1] M. Casas, R. M. Badia, and J. Labarta. Automatic phase detection and structure extraction of MPI applications. *International Journal of High Performance Computing Applications*, 24(3):335–360, 2010.

[2] Z. Diao, Z. Li, S. Wang, Y. Ding, A. Panchula, E. Chen, L.-C. Wang, and Y. Huai. Spin-transfer torque switching in magnetic tunnel junctions and spin-transfer torque random access memory. *Journal of Physics: Condensed Matter*, 19(16):165209, 2007.

[3] J. Gonzalez, J. Gimenez, M. Casas, M. Moreto, A. Ramirez, J. Labarta, and M. Valero. Simulating whole supercomputer applications. *Micro, IEEE*, 31(3):32–45, 2011.

[4] R. Iyer, H. Wang, and L. Bhuyan. Design and analysis of static memory management policies for CC-NUMA multiprocessors. *College Station, TX, USA, Tech. Rep*, 1998.

[5] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Trans. Comput. Syst.*, 10(4):338–359, Nov. 1992.

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

[7] G. Loh. 3D-Stacked Memory Architectures for Multi-core Processors. In *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, pages 453–464, 2008.

[8] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating system support for NVM+DRAM hybrid main memory. In *Proceedings of the 12th conference on Hot topics in operating systems*, HotOS'09, pages 14–14, Berkeley, CA, USA, 2009. USENIX Association.

[9] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 42(6):89–100, 2007.

[10] M. Pavlovic, Y. Etsion, and A. Ramirez. On the memory system requirements of future scientific applications: Four case-studies. *IEEE Workload Characterization Symposium*, 0:159–170, 2011.

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

[12] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page placement in hybrid memory systems. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 85–95, New York, NY, USA, 2011. ACM.

[13] A. Rico, F. Cabarcas, A. Quesada, M. Pavlovic, A. J. Vega, C. Villavieja, Y. Etsion, and A. Ramirez. Scalable simulation of decoupled accelerator architectures. *Universitat Politecnica de Catalunya, Tech. Rep. UPC-DACRR-2010-14*, 2010.

[14] A. Rico, F. Cabarcas, C. Villavieja, M. Pavlovic, A. Vega, Y. Etsion, A. Ramirez, and M. Valero. On the simulation of large-scale architectures using multiple application abstraction levels. *ACM Trans. Archit. Code Optim.*, 8(4):36:1–36:20, Jan. 2012.

[15] A. D. Simpson, M. Bull, and J. Hill. *Identification and Categorisation of Applications and Initial Benchmarks Suite*, 2008. www.prace-project.eu.

[16] D. Vicente and J. Bartolome. BSC-CNS Research and Supercomputing Resources. In *High Performance Computing on Vector Systems 2009*, pages 23–30. Springer, 2010.

[17] C. Villavieja, Y. Etsion, A. Ramirez, and N. Navarro. FELI: HW/SW support for on-chip distributed shared memory in multicores. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par'11, pages 282–294, Berlin, Heidelberg, 2011. Springer-Verlag.

[18] R. Waser, R. Dittmann, G. Staikov, and K. Szot. Redox-Based Resistive Switching Memories–Nanoionic Mechanisms, Prospects, and Challenges. *Advanced Materials*, 21(25-26):2632–2663, 2009.

[19] Y. Xie. Modeling, Architecture, and Applications for Emerging Memory Technologies. *Design Test of Computers, IEEE*, 28(1):44–51, 2011.