# ProFess: A Probabilistic Hybrid Main Memory Management Framework for High Performance and Fairness

Dmitry Knyaginin
Chalmers University of Technology
Gothenburg, Sweden
d.knyaginin@gmail.com

Vassilis Papaefstathiou
FORTH-ICS
Heraklion, Crete, Greece
papaef@ics.forth.gr

Per Stenstrom
Chalmers University of Technology
Gothenburg, Sweden
per.stenstrom@chalmers.se

## ABSTRACT

Non-Volatile Memory (NVM) technologies enable cost-effective *hybrid* main memories with two partitions: *M1* (DRAM) and slower but larger *M2* (NVM). This paper considers a flat migrating organization of hybrid memories. A challenging and open issue of managing such memories is to allocate M1 among co-running programs such that high fairness is achieved at the same time as high performance. This paper introduces *ProFess*: a **Pro**babilistic hybrid main memory management **F**ramework for high performance and fairn**ess**. It comprises: i) a *Relative-Slowdown Monitor (RSM)* that enables fair management by indicating which program suffers *the most* from competition for M1; and ii) a *probabilistic Migration-Decision Mechanism (MDM)* that unlocks high performance by realizing cost-benefit analysis that is *individual* for each pair of data blocks considered for migration. Within ProFess, RSM guides MDM towards high fairness. We show that for the multiprogrammed workloads evaluated, ProFess improves fairness by 15% (avg.; up to 29%), compared to the state-of-the-art, while outperforming it by 12% (avg.; up to 29%).

***Keywords:*** hybrid main memory; flat migrating organization; fairness; performance; hardware

## 1. INTRODUCTION

Non-Volatile Memory (NVM) technologies like 3D Xpoint [1] promise higher bit densities and lower costs per bit than DRAM. Thus, they enable cost-effective *hybrid* main memories with two partitions: *M1* (DRAM) and *M2* (NVM), where M2 is slower but larger than M1.

Two hybrid-memory topologies have been considered in the literature: in *flat* memories the processor can directly access both M1 and M2 [2, 3, 4, 5, 6, 7, 8], but in *hierarchical* memories it can access M2 only via M1 [9, 10, 11]. Regardless of the topology, upon a promotion of data (from M2 to M1) *migrating* memories move the data [9, 2, 11, 4, 5, 6, 7, 8], but *replicating* memories copy it thereby limiting the total capacity visible to the Operating System (OS) to that of M2 [10, 3].

A lucrative way to build hybrid main memory is to use the flat topology and to place NVM close to the processor, next to DRAM on the memory channels. For instance, a channel in Intel Purley [12] can accommodate one DRAM module and one 3D Xpoint module. Still significant DRAM module capacities (compared to those of NVM) motivate migrating hybrid memories. Large numbers of entries for translating original addresses to actual addresses in M1 and M2 motivate organizations that store the entries in M1 [4, 5, 6, 7].

Management of flat migrating hybrid memories has two major challenges. First, relative to M2, M1 is a limited shared resource, and co-running programs compete for it. If a program fails to occupy a share of M1 large enough for its needs, it may suffer an excessive slowdown, that would hurt fairness (the higher the max slowdown among the co-running programs, the lower the fairness of the system [13, 14]). Thus, to maximize system fairness, it is important to allocate M1 such that the max slowdown is minimized. Second, it is important to perform only promotions that *clearly benefit performance*, i.e., promotions where the benefit exceeds the overhead and the penalty of respective demotions. This implies that, to maximize performance, migration decisions should be based on cost-benefit analysis that is *individual* for each pair of data blocks in M1 and M2 considered for migration.

The existing schemes [4, 5, 6, 7] have two major limitations. First, they suffer from fairness issues due to ignoring individual program slowdowns: the schemes offer no provision to assess the slowdowns and to maintain fairness. Second, they compromise performance due to ignoring individual cost-benefit analysis [4, 5, 6, 7] and due to using global thresholds on the number of accesses to a block in M2 before promoting it [4, 5, 6] (such global thresholds degrade cost-benefit analysis to a one-size-fits-all heuristic).

This paper proposes *ProFess*: a **Pro**babilistic hybrid main memory management **F**ramework for high performance and fairn**ess**. ProFess comprises: i) a *Relative-Slowdown Monitor (RSM)* that enables fair management by indicating which program suffers the most from the competition for M1; and ii) a *probabilistic Migration-Decision Mechanism (MDM)* that attains high performance by realizing individual cost-benefit analysis.

RSM is *the first* among the existing schemes [4, 5, 6, 7] to improve fairness in flat migrating hybrid memories. It leverages our intuition that in hybrid memories the competition for M1 is *the major* performance factor, i.e., that the max slowdown can be reduced by allocating more M1 to a program that suffers the most from

2378-203X/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCA.2018.00022

the competition. MDM is *the first* among the existing schemes [4, 5, 6, 7] to realize cost-benefit analysis that is individual for each pair of blocks in M1 and M2. It leverages our insight that *expected* numbers of accesses to each data block can be predicted based on a *block attribute*, proposed to distinguish blocks, and statistics of the numbers of accesses per attribute value. To achieve high fairness, ProFess guides MDM's migration decisions such that they help a program indicated by RSM.

This paper addresses the major limitations of the existing schemes [4, 5, 6, 7] by making the following contributions. For the first time in flat migrating hybrid memories, it proposes: 1) an approach to improve fairness by reducing the max slowdown among the co-running programs, and 2) a new approach to making migration decisions that is probabilistic and realizes individual cost-benefit analysis. Lastly, the paper integrates the two contributions into a framework. We show that for the multiprogrammed workloads evaluated, ProFess improves fairness by 15% (avg.; up to 29%), compared to the state-of-the-art [5]. At the same time, ProFess outperforms it by 12% (avg.; up to 29%). We manage to improve fairness and performance at the same time, since the latter is measured as weighted speedup [15].

The paper is organized as follows. Section 2 provides background and motivation, Section 3 presents ProFess, Section 4 describes the experimental setup, and Section 5 the results. Lastly, Section 6 discusses the related work, and Section 7 concludes the paper.

# 2. BACKGROUND AND MOTIVATION

## 2.1 Memory Technologies

NVM such as phase-change memory [16] and 3D Xpoint [1] scale better than DRAM, promising higher bit densities and lower costs per bit. However, they are multiple times slower than DRAM, and thus cannot replace it completely without significant performance losses. On the other hand, combining DRAM and NVM increases the total capacity, enabling cost-effective main memories that still have high performance [10]. Even NAND Flash, although several orders of magnitude slower than DRAM, can be employed for memory extension [11, 17, 18]. This paper considers NVM similar to the expected 3D Xpoint by being eight times denser but one order of magnitude slower than DRAM.

## 2.2 Flat Migrating Hybrid Memory

We consider hybrid memory with one DRAM module and one NVM module per channel, like in Intel Purley [12]. We assume that M1 and M2 modules have the same number of memory devices. Thus, according to the assumed NVM density, M2 modules are eight times denser than M1 modules. Since the capacity of M1 is a significant fraction of the total capacity, we organize this hybrid memory as migrating.

Prior work [4, 5, 6, 7] comprehensively motivates that hybrid memory has to be managed by hardware, transparently to the OS, which is important to enable management at sub-page granularities, avoid high per-migration overheads (such as TLB and cache flushes), and thereby retain *responsiveness to working-set changes*. OS-based management [8] can be feasible for a different goal, e.g.: given that initially the *entire* footprint is in DRAM, dynamically identify and move to NVM huge (2-MB) pages accessed only about 30K times per second, to stay within 3% of the DRAM-only performance. Such coarse-grain management implies slow responsiveness to working-set changes, and effectively requires that about 50% of the footprint remains in DRAM [8]. On the contrary, we employ hardware-based management to implement aggressive, relatively low-overhead migrations, thus retaining responsiveness necessary to maximize performance and fairness.

**Table 1: Flat Migrating Organizations**

| | M1:M2 Capacity Ratio | M1-M2 Addr. Mapping | Block Size | Swap Type |
|---|---|---|---|---|
| CAMEO [4] | 1:3 | Direct-mapped | 64B | Fast |
| PoM [5] | Config. (1:4, 1:8) | Direct-mapped | 2KB | Fast |
| SILC-FM [6] | Config. (1:4) | Set-assoc. | 64B-2KB | Slow |
| MemPod [7] | Config. (1:8) | Fully-assoc. | 2KB | Fast |

*Capacity ratio in parentheses = ratio used for main evaluation in respective work. Swap type is according to definition in PoM [5]*

Each request arriving to the Memory Controller (MC) requires a translation of its *original* physical address (originally allocated by the OS) to the *actual* one (changed after a migration), and the MC manages the respective address-translation entries. Such entries total dozens of megabytes, and so they have to be stored in M1 [4, 5, 6, 7]. To minimize the size of each entry and to simplify their own addressing in M1, possible M1-M2 address mappings can be restricted, such that only specific addresses in M2 can map to a given address in M1 [4, 5]. This way, only a few bits of the original address have to be translated, and the remaining bits, common to the possible actual addresses, uniquely address the respective translation entry in M1.

## 2.3 Baseline Organization

Table 1 summarizes the existing flat migrating organizations. CAMEO [4] is optimized for memories where i) M1 is 1/4-th of the total capacity, ii) one of three fixed physical addresses in M2 can map to a single fixed physical address in M1 (forming *swap groups* of four blocks), iii) data are migrated at the 64-B block granularity, and iv) the type of *swaps* (promotions causing demotions) is *fast* (more than two blocks can be remapped within a swap group). Unlike CAMEO, PoM [5] can be configured to support M1:M2 capacity ratios other than just 1:3, which makes it practical in our study, where the ratio is 1:8. SILC-FM [6] relaxes the M1-M2 address mapping, making it set-associative, where an M2 block from one swap group can be swapped with an M1 block in another swap group. In addition, SILC-FM supports sub-block interleaving [6], i.e., swap-block sizes variable at the 64-B granularity. However, this relaxation comes at a cost of swaps being *slow* (the original mapping in a swap group must be restored before each swap). MemPod [7] further relaxes the M1-M2 address mapping by

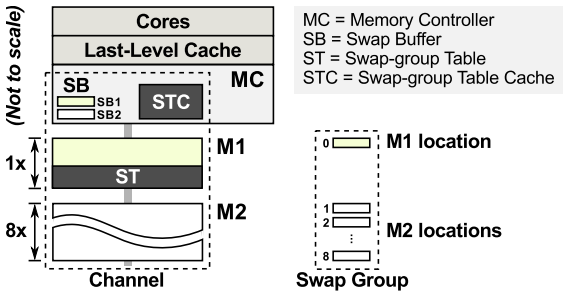


**Figure 1: Baseline flat migrating organization**

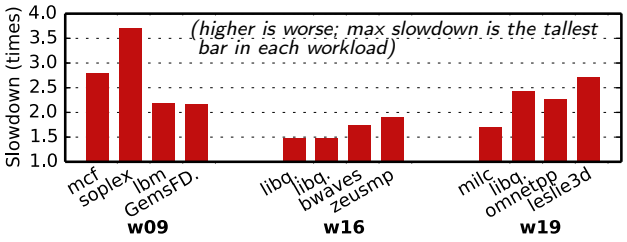


**Figure 2: Slowdowns under PoM management**

making it fully-associative, at a cost of a dramatic increase of the address-translation storage overhead.

In this paper we choose the PoM organization [5] as a baseline. We prefer it to the SILC-FM [6] and MemPod [7] organizations in order to focus on the quality of migration decisions, motivated as follows. For each pair of blocks considered for a swap, a migration algorithm has to decide which block to award M1. By excluding the M1-M2 address-mapping associativity of SILC-FM and MemPod, we restrict the number of such pairs, and thus emphasize the effect of each migration decision. In addition, by using a single organization for comparing different algorithms, we make the comparison fair. Note that, fundamentally, *M1-M2 address-mapping relaxations are orthogonal to migration algorithms*, since the possible address mappings solely define candidates for a swap, and a migration algorithm merely decides whether to swap the blocks or not.

Figure 1 shows a baseline system where each channel has one M1 module and one M2 module (only one channel is shown, though the system has multiple such channels). Data are migrated at the 2-KB granularity [5]. To support migrations, the MC employs two 2-KB *Swap Buffers*, *SB1* and *SB2*. All memory locations are organized into swap groups of nine fixed physical locations: one in M1 and eight in M2 [5]. Thus, only $\lceil log_2 9 \rceil = 4$ bits of each original address have to be translated. Address translations are stored in M1 in a *Swap-group Table (ST)*. Recently accessed ST entries are stored on-chip in a *Swap-group Table Cache (STC)*.

## 2.4 The Fairness Problem

Compared to M2, M1 is a limited shared resource. A program that fails to obtain enough M1 for its needs may experience an excessive slowdown, given by

$$sdn = IPC_{SP} / IPC_{MP}, \quad (1)$$

where $IPC_{SP}$ is the program's Instructions Per Cycle (IPC) when it runs alone (uncontended IPC) and $IPC_{MP}$ is that when it runs within the workload (IPC under contention). For fair execution, M1 should be allocated such that the max slowdown across the co-running programs is minimized.

However, the existing schemes [4, 5, 6, 7] strive to maximize system performance disregarding the performance of individual programs. This inevitably leads to excessive slowdowns for some programs. Figure 2 shows such slowdowns in three four-program workloads under the PoM management [5] (the experimental setup is described in Section 4). For instance, in workload w09 the slowdown of soplex is 3.7, but that of lbm and GemsFDTD is just about 2.2. Likewise, zeusmp in workload w16 and leslie3d in w19 experience excessive slowdowns. The other existing schemes [4, 6, 7] disregard individual program slowdowns, too, and so they suffer from the same fairness problem.

**Table 2: Migration Algorithms**

| | Cost-Benefit Analysis | Migration Condition |
|---|---|---|
| CAMEO [4] | No | Global threshold of 1 access |
| PoM [5] | Yes, global | Global adaptive threshold (1, 6, 18, or 48 accesses) or prohibit migrations |
| SILC-FM [6] | No | Global threshold of 1 access; locked in M1 if aging access counter > 50 |
| MemPod [7] | No | Majority Element Algorithm (MEA) [19], up to 64 migrations every 50$\mu$s |

## 2.5 The Performance Problem

For high performance, the benefit of each promotion must exceed the overhead of the swap. This implies that a cost-benefit analysis has to be performed *individually* for each pair of blocks considered for a swap.

However, the existing migration algorithms [4, 5, 6, 7], summarized in Table 2, do not perform individual cost-benefit analysis, thus compromising performance. In addition, a lack of *any* cost-benefit analysis makes an algorithm less adaptive. For instance, we experimentally find that in our system the MemPod algorithm [7], *underperforms* compared to the PoM algorithm [5], for the best MemPod configuration we could find (Section 4 describes the setup). Specifically, MemPod's average main memory access time—the metric [7] preferred by the authors of MemPod—is longer, on average, by 19% and 18% in our single- and multi-program experiments, respectively, compared to PoM. Both algorithms are originally proposed for memories where M1 is on-chip DRAM and M2 is off-chip DRAM. Our system employs different technologies, and PoM adapts to them better than MemPod, thanks to its *global* cost-benefit analysis, that takes into account technology characteristics.

The problem with global thresholds is that they simplify cost-benefit analysis to a one-size-fits-all heuristic. For instance, the CAMEO algorithm [4] is optimized for 64-B blocks and presumes that all swaps are justified after one access (it uses a global threshold of 1). Consider a pattern where both blocks are accessed repeatedly,

one after another. CAMEO would swap blocks after each access, although a higher performance would be achieved without any swaps.

PoM [5] is more adaptive, since it chooses one of four global thresholds (Table 2) based on their respective *global benefit*, estimated per epoch. If none of the thresholds yields a positive benefit estimate, PoM prohibits swaps for the next epoch. The major limitation of PoM is conceptually the same as that of CAMEO: it degrades individual cost-benefit analysis to a one-size-fits-all heuristic. Consider a global threshold of 48, a block in M1 that is not accessed, and a block in M2 that is accessed 49 times. PoM would promote the block in M2 after 48 accesses, although promoting it after the first access would yield a higher performance.

The SILC-FM algorithm [6] uses global thresholds, does not perform cost-benefit analysis, and thus experiences the same performance problem. Although MemPod [7] does not use global thresholds, it suffers from a lack of cost-benefit analysis, as we discuss above.

# 3. FAIR AND HIGH-PERFORMANCE HYBRID MEMORY MANAGEMENT

This section presents ProFess, a **Pro**babilistic hybrid main memory management **F**ramework for high performance and fairn**ess**, comprising a Relative-Slowdown Monitor (RSM) and a Migration-Decision Mechanism (MDM). ProFess uses RSM to guide MDM to address the fairness problem of Section 2.4, while MDM tackles the performance problem of Section 2.5. Next, Section 3.1 presents the intuition and a practical implementation of RSM; Section 3.2 those of MDM; and Section 3.3 the integration of RSM and MDM.

## 3.1 Relative-Slowdown Monitor

Fair management throughout the execution of a multiprogrammed workload requires dynamic estimation of each program's slowdown, given by (1). The dynamic estimation of each program's stand-alone performance ($IPC_{SP}$ in (1))—i.e., estimation of each program's performance *as if* it was running alone when it actually co-runs with the other programs—is a challenging and open issue in hybrid memories.

We propose RSM, a Relative-Slowdown Monitor based on two key insights. First, the large speed gap between M1 and M2 (Section 2.1) makes the competition for M1 among co-running programs *the major* performance factor. Thus, the competition can *proxy* slowdown. However, since the slowdown can be caused by multiple additional factors (such as bandwidth contention), the competition for M1 cannot accurately estimate it. Thus, we propose to merely indicate which program suffers *the most* from the competition, using relative numbers. Our intuition is that allocating more M1 to that program would increase fairness.

Second, swap groups can be further grouped into *regions*, such that one region per program is *private* (only data of that program can be mapped to the region), and the other regions are shared. Then, each program's *behavior*—the number of served requests and swaps—in its private region is unaffected by the competition for M1. We propose to assume this behavior uncontended and to proxy by it the program's stand-alone performance. Likewise, we propose to proxy the program's performance under contention by the program's behavior in the shared regions. The ratio of the two proxies, like in (1), can then proxy the program's slowdown *caused specifically by the competition for M1* (not to be confused with the actual slowdown estimated by (1)).

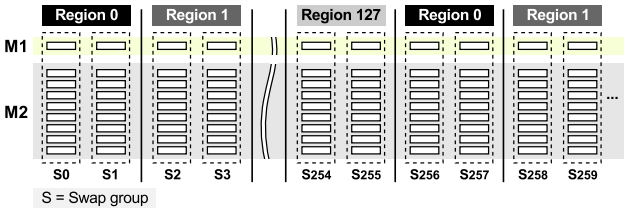


**Figure 3: Interleaved division into 128 regions**

Since we define a program's behavior by 1) the number of served requests and 2) the number of swaps, we propose to use two separate *Slowdown Factors*, $SF_A$ and $SF_B$, where $SF_A$ indicates the competition for M1 *in terms of served requests*, and $SF_B$ *in terms of swaps* (we clarify these definitions in Section 3.1.2). That is, $SF_A$ and $SF_B$ indicate, each in its own way, how much a program suffers from the competition for M1, and their *relative* values can indicate which of the programs suffers *the most*. To increase fairness, we propose to help that program to obtain more M1 by guiding migration decisions. The rest of this section details RSM.

### 3.1.1 Private and Shared Regions

We divide hybrid main memory into *interleaved* regions along the swap groups, as shown in Figure 3 for 128 regions (from left to right: Region 0, 1, . . . , 127, 0, 1, . . . ). We use 4-KB OS pages and 2-KB swap blocks, thus each page maps to two consecutive swap groups, which must map to the same region. For instance, in Figure 3 swap groups S0, S1, S256, S257, and so on map to Region 0, and swap groups S2, S3, S258, S259, and so on map to Region 1. Such interleaving reduces the non-uniformity of access distribution across the regions.

Next, we propose to dedicate one region per program to form *private* regions. To address the case when competition is among multi-threaded programs, we dedicate the same region to all threads of a multi-threaded program. The case when competition is among threads of the same program requires additional studies, where such factors like the programming model, sharing degree, and the threads' criticality could be considered.

The total number of regions should be large compared to the number of concurrently executing programs, so that the fraction of private regions is small. The OS must keep track of free M1 and M2 physical page frames per region and to allocate frames of the private regions to their respective programs only. The MC can decode the region type from the actual physical address of each request and a region map defined by the OS. Thus, unlike the existing schemes [4, 5, 6, 7], that suffer from

**Table 3: Per-Program Set of RSM Counters**

| | |
|---|---|
| $num\_Req\_M1\_P$ | Req. served from M1 of the private region |
| $num\_Req\_Total\_P$ | Req. served from M1 and M2 of the private region |
| $num\_Req\_M1\_S$ | Req. served from M1 of the shared regions |
| $num\_Req\_Total\_S$ | Req. served from M1 and M2 of the shared regions |
| $num\_Swap\_Self$ | Swaps where both blocks belong to the program |
| $num\_Swap\_Total$ | Swaps where at least one block belongs to the program, regardless which program triggers the swap |

**Table 4: Estimates of Sampling Accuracy**

| | Mean $\hat{\sigma}_{req}$ (%) | | | $\hat{\sigma}_{raw\_SF_A}$ (%) | | | $\hat{\sigma}_{avg\_SF_A}$ (%) | | |
|---|---|---|---|---|---|---|---|---|---|
| | 64K | 128K | 256K | 64K | 128K | 256K | 64K | 128K | 256K |
| bwaves | 36 | 26 | 18 | 3 | 2 | 1 | 0.5 | 0.3 | 0.2 |
| milc | 27 | 20 | 15 | 21 | 13 | 10 | 5.1 | 3.3 | 2.7 |
| omnetpp | 15 | 12 | 10 | 6 | 5 | 4 | 2.1 | 1.6 | 1.4 |

the fairness problem of Section 2.4, our proposal enables fair management with this OS support. Note that swaps are still transparent to the OS.

Our proposal is independent of which core(s) a program runs on, so programs can migrate among the cores. To make the discussion concise, we assume that programs are single-threaded and pinned to cores, allowing us to refer to programs and cores interchangeably.

### 3.1.2 Slowdown Factors $SF_A$ and $SF_B$

To assess how a program suffers from the competition for M1, we need to compare the program's behavior in its private region to its behavior in the shared regions (as described in Section 3.1). The following discussion is per program (per core), and so we do not show the respective ID. Recall that all threads of a multi-threaded program appear to RSM as a single program.

To monitor a program's behavior, we propose *RSM counters* (Table 3) assigned per program via a hardware lookup table (the program ID indicates which set of RSM counters to update upon a served request or a swap; this way we support context switching). Using the counters, we periodically compute slowdown factors $SF_A$ and $SF_B$:

$$SF_A = \left(\frac{num\_Req\_M1\_P}{num\_Req\_Total\_P}\right) \Big/ \left(\frac{num\_Req\_M1\_S}{num\_Req\_Total\_S}\right), \quad (2)$$

$$SF_B = 1 \Big/ \left(\frac{num\_Swap\_Self}{num\_Swap\_Total}\right). \quad (3)$$

$SF_A$ indicates how a program suffers from the competition for M1 in terms of served requests, by dividing the fraction of requests served from M1 of the private region (no competition) over that of the shared regions. The intuition is that a higher competition would decrease the denominator, thus increasing the value of $SF_A$.

$SF_B$ indicates competition in terms of swaps, i.e., it describes how different is a program's behavior in terms of swaps in the shared regions from that in the program's private region. $SF_B$ is based on our intuition that the smaller the fraction of swaps where *both blocks belong to the program* (i.e., where the program "swaps itself"), the higher the competition. In the private region all blocks belong to the same program, and so this fraction is always 1—hence the numerator in (3)—and we do not count swaps there. An $SF_B = 1$ means no competition, i.e., that the program swaps only its own data, like in its private region. A large $SF_B$ indicates high competition, i.e., that the majority of swaps involve blocks of different programs, which diverges from the behavior in the private region and is thus undesired.

Note the similarity of each of (2) and (3) to (1), where the numerator is the uncontended performance, and the denominator is the performance under contention. Recall, however, that $SF_A$ and $SF_B$ cannot precisely estimate the actual slowdown of (1). Instead, they proxy, each in its own way, *the slowdown due to the competition for M1*. Thus, the end purpose of $SF_A$ and $SF_B$ is merely to indicate which program suffers the most from the competition. Then, fairness can be increased by helping that program to obtain more M1. In Section 3.3, we identify three cases that depend on the *relative* values of $SF_A$ and $SF_B$ of two programs participating in a swap, and propose a strategy to guide migration decisions towards high fairness.

### 3.1.3 Sampling Accuracy

We recompute $SF_A$ and $SF_B$ independently for each program at the end of every *sampling period*, resetting the RSM counters (Table 3) at the beginning of the period. Since the private regions represent a small fraction of the total capacity, the request counters in (2) can be noisy. (Since we do not count swaps in the private regions, we do not discuss swaps here.) For reliable estimates, sampling error should be small, requiring an appropriate choice of a sampling-period duration, denoted by $M_{samp}$ and measured in served requests. Ideally, $M_{samp}$ should be such that when a program runs alone, the mean $SF_A$ estimate across all sampling periods throughout execution is 1 with zero variance.

Let us consider a simplified analytic model. Let $N$ denote the number of regions and $M$ the total number of memory accesses. For independent accesses, the probability of each access going to a region is $1/N$ for all regions. The distribution of the number of accesses per region after $M$ accesses is Multinomial (the problem is identical to rolling an $N$-sided die $M$ times), and the standard deviation is given by

$$\sigma = \sqrt{M \cdot \frac{1}{N}\left(1 - \frac{1}{N}\right)} = \frac{\sqrt{M(N-1)}}{N}. \quad (4)$$

For instance, for $N = 128$ regions and $M = 2^{17}$ accesses, the standard deviation is about 32 accesses per region, i.e., about $32/(M/N) = 32/1024 \approx 3\%$. This model assumes a uniform access distribution across the regions and is thus idealized, yielding the lowest possible $\sigma$ per $M$. In a real system, the standard deviation would be greater due to non-uniformity of the access distribution.

Table 4 shows *experimental* estimates of sampling accuracy across all sampling periods throughout execution for $M_{samp}$ of 64K, 128K, and 256K requests for selected programs running alone. (The setup is described in Section 4; programs from Section 4 not shown in Table 4 provide no additional insights.) Columns 1-3 show the mean of $\hat{\sigma}_{req}$, which is the standard deviation of the

number of requests served per region during $M_{samp}$. For instance, for bwaves doubling $M_{samp}$ to 128K reduces the mean $\hat{\sigma}_{req}$ from 36% to 26%. Columns 4-6 and 7-9 show the standard deviation of raw ($\hat{\sigma}_{raw\_SF_A}$) and averaged ($\hat{\sigma}_{avg\_SF_A}$) $SF_A$ estimates, respectively. The raw estimates use the counter values in (2) as-is, while the averaged estimates use them after simple exponential smoothing with a smoothing parameter $\alpha = 0.125$. The means of the raw and averaged $SF_A$ are ~1, and we omit them for brevity. Table 4 shows that the averaging significantly reduces the standard deviation: e.g., milc at 128K has $\hat{\sigma}_{raw\_SF_A}$ of 13% but $\hat{\sigma}_{avg\_SF_A}$ of just 3.3%.

In general, non-uniformity of access distribution depends on the access pattern and the allocation of physical page frames. We propose to choose $M_{samp}$ that gives small standard deviation according to (4) and to apply simple exponential smoothing to the RSM counters (Table 3). To avoid zeros, we increment by one each counter before adding it to the respective average.

Next, Section 3.2 presents our Migration-Decision Mechanism (MDM), that maximizes performance regardless of fairness. Then, Section 3.3 explains how RSM's $SF_A$ and $SF_B$ guide MDM towards high fairness.

## 3.2 Migration-Decision Mechanism

We propose a conceptually new, *probabilistic* Migration-Decision Mechanism (MDM), that realizes *individual* cost-benefit analysis for each pair of blocks considered for a swap. The cost is the swap overhead, estimated using the number of memory requests required to swap the blocks and the characteristics of M1 and M2. The benefit is estimated by *statistically* predicting the number of *remaining* accesses to the block in M2 and that to the block in M1, and then subtracting the latter from the former. A swap is performed only if the benefit is greater than the cost. To enable predictions, MDM keeps an access counter per block and gathers statistics about the numbers of accesses. Then, the predicted number of remaining accesses to a block is a probabilistically computed expected number of accesses to the block minus its current access count. To make this approach practical, MDM employs access counters only for actively accessed blocks.

MDM is based on two insights. The first and key one is that we can statistically predict expected numbers of accesses for each block of each program separately. Blocks of each program can be identified by an *attribute* with a small number of values. Thus, statistics collection and access-number prediction can be performed per program per attribute value.

The second insight is that the Swap-group Table Cache (STC, Figure 1) can function as a temporal filter to identify periods when blocks are *actively* accessed: while a block's Swap-group Table (ST) entry is resident in the STC, the block is *likely* being accessed (there is uncertainty since the ST entry might reside in the STC due to accesses to other blocks in the same swap group). Then, a block's temporal access pattern can be represented by the numbers of accesses to the block counted during its ST-entry residency in the STC.

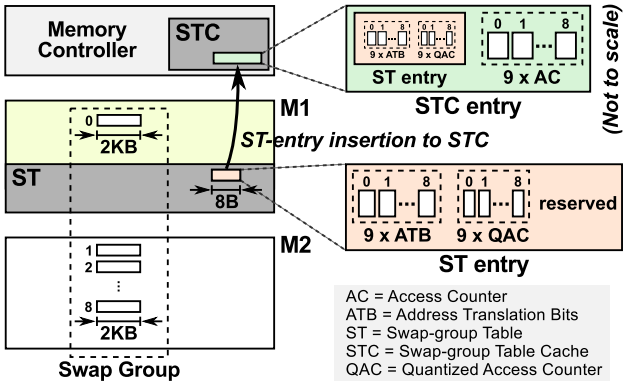


**Figure 4: ST entry and STC organization**

**Table 5: Quantized Access-Counter Values**

| Value | Meaning | Value | Meaning |
|---|---|---|---|
| 0 | Previously unseen block (default) | 2 | 8-31 accesses |
| 1 | 1-7 accesses | 3 | 32 or more accesses |

Combining the two insights, we propose to define a block attribute as a *quantized* number of accesses to the block, counted during the last residency of its ST entry in the STC. Based on the attribute value, we can predict an expected number of accesses to the block next time its ST entry gets cached in the STC.

Further, by using the STC as a temporal filter, we can limit the number of ST entries for which accurate state—such as one access counter per block—has to be maintained. Specifically, the number of such ST entries can be limited only to ST entries resident in the STC. This way, we can keep accurate state inexpensively.

The conceptual difference of MDM from the existing schemes [4, 5, 6, 7] is that it makes migration decisions based on a *predicted* number of *remaining* accesses to each block. This way, the probability of a promotion that *clearly benefits performance* is the *highest* at a *first access* to a block in M2, i.e., when the block's number of *already served* accesses is the *smallest* and, respectively, the number of *remaining* accesses is the *greatest*. Such timely and accurate promotions are less likely under the existing schemes, since they either consider promotions beneficial based on solely the number of *already served* accesses [5, 7] or promote *all* blocks after the first access [4, 6], experiencing the performance problem of Section 2.5. As a result, swaps performed by MDM are likely to have a greater benefit for overall performance. The rest of this section details MDM.

### 3.2.1 ST Entry and STC Organization

We propose to use one access counter per block (swap-group location) for ST entries resident in the STC. A swap group comprises nine locations, and its ST entry contains address-translation bits for each of them, as shown in Figure 4. The access counters belong to the STC and get reset to 0 at ST-entry *insertion* to the STC. The ST entry itself does not store the counter values, but rather their quantized values, that get up-

**Table 6: Per-Program Set of MDM Counters**

| | |
|---|---|
| $accum\_cnt(q_E)$ | Accumulator for access counts of blocks with $q_E$ |
| $num\_q\_sum_I(q_E)$ | Counter of all transitions to $q_E$ (regardless $q_I$) |
| $num\_q(q_I, q_E)$ | Counter of blocks with $q_I$ transitioned to $q_E$ |
| $num\_q\_sum_E(q_I)$ | Counter of all transitions from $q_I$ (regardless $q_E$) |

dated at ST-entry *eviction* from the STC. Table 5 explains the Quantized Access-Counter (QAC) values. ST entries are initialized with the default QAC value of 0. If a block's access count is 0 at ST-entry eviction, the MC does not update the block's QAC value.

We experimentally observe that MDM incurs no significant extra memory traffic due to updating QAC at ST-entry evictions. This is because: i) each PoM's ST entry also includes Address Translation Bits (ATB, Figure 4) and in addition one competing counter [5]; and ii) the STC hit rates in PoM and MDM are similar and high, averaging 94% for our multiprogrammed workloads (Section 4.2). Thus, by the time of an ST-entry eviction, it is typical that ATB change (due to swaps), requiring a read-modify-write regardless whether the competing counter in PoM or QAC in MDM change.

#### 3.2.2 Prediction of Expected Number of Accesses

Each block is identified by its program ID (core ID) and QAC value. The following discussion is per program (per core), and so we do not show the respective ID. Just like in Section 3.1, all threads of a multithreaded program appear to MDM as a single program.

Let us denote by $q_I$ the QAC value at insertion of the swap group's ST entry to the STC, by $q_E$ that at eviction, by $num\_q_I$ the number of $q_I$ values, and by $num\_q_E$ the number of valid $q_E$ values. According to Table 5, $num\_q_I = 4$. Since QAC values are updated only for blocks with non-zero access counts, $q_E = 0$ is invalid. Thus, $num\_q_E = num\_q_I - 1 = 3$.

Next, let us denote by $avg\_cnt(q_E)$ an average access count per $q_E$, and by $P(q_E \mid q_I)$ the probability of $q_I$ transitioning to $q_E$. Table 6 defines a set of *MDM counters*, assigned per program via a hardware lookup table. Then, for a block with $q_I$ we propose to estimate an expected number of accesses $exp\_cnt(q_I)$ by

$$exp\_cnt(q_I) = \sum_{q_E=1}^{num\_q_E} avg\_cnt(q_E) \cdot P(q_E \mid q_I), \qquad (5)$$

where

$$avg\_cnt(q_E) = \frac{accum\_cnt(q_E)}{num\_q\_sum_I(q_E)} \qquad (6)$$

and

$$P(q_E \mid q_I) = \frac{num\_q(q_I, q_E) + 1}{num\_q\_sum_E(q_I) + num\_q_E}. \qquad (7)$$

Table 6 describes the counters in (6) and (7). In (7), the constants of 1 in the numerator and $num\_q_E = 3$ in the denominator implement Laplace smoothing.

The MC updates the MDM counters at each ST-entry eviction for each block with a non-zero access count. However, the MC updates $avg\_cnt(q_E)$ and $P(q_E \mid q_I)$ *periodically*, in phases, where an observation phase (no updates) is followed by an estimation phase (updates at equal intervals). We discuss their periodicity at the end of Section 4.1, chosen such that the multiply and divide operations are relatively infrequent (we experimentally observe that the intervals between re-calculations average ~100K cycles for our multiprogrammed workloads). Note that the updates are not on the critical path.

In terms of hardware, the MC needs 22 counters: $num\_q_E \times 2 = 6$ counters to compute $avg\_cnt(q_E)$ and $num\_q_I \times num\_q_E + num\_q_I = 4 \times 3 + 4 = 16$ counters to compute $P(q_E \mid q_I)$. Updates of $avg\_cnt(q_E)$ and $P(q_E \mid q_I)$ trigger updates of $exp\_cnt(q_I)$, and the values are registered between updates, requiring $num\_q_E + num\_q_I \times num\_q_E + num\_q_I = 3 + 4 \times 3 + 4 = 19$ registers.

#### 3.2.3 Migration Decisions

Upon an access to a block, the MC increments its access counter in the STC. If the block is in M2, the MC assesses the benefit of promoting it using a cost-benefit analysis (not on the critical path). First, the MC estimates a *remaining* number of accesses to the block by

$$rem\_cnt_{M2} = exp\_cnt(q_I) - curr\_cnt, \qquad (8)$$

where $q_I$ is the block's QAC value at insertion of its ST entry to the STC, $exp\_cnt(q_I)$ is an expected number of accesses precomputed according to (5), and $curr\_cnt$ is the block's current access-counter value. Then, the top-level condition is

$$rem\_cnt_{M2} \geq min\_benefit,$$

where $min\_benefit$ is the least number of *remaining* accesses that justifies a promotion (its value depends on the characteristics of M1 and M2; Section 4 discusses how to compute it). If the condition is false, there is no benefit to promote the block. Otherwise, the MC schedules a promotion if: a) M1 is vacant (not an expected case); or b) M1 is occupied, but has not been accessed (its access counter is 0), and *some other* block in the same swap group has been accessed (which hints that the block in M1 is unlikely to be accessed soon); or c) M1 is occupied, its access counter is not zero, and c.i) $rem\_cnt_{M1} \leq 0$; or c.ii) $rem\_cnt_{M1} > 0$ and

$$rem\_cnt_{M2} - rem\_cnt_{M1} \geq min\_benefit,$$

where $rem\_cnt_{M1}$ is a remaining number of accesses to the block in M1 (predicted just like in (8)). If the condition is false, the swap has no benefit. Thus, the key net difference of MDM from the existing schemes [4, 5, 6, 7] is that it *predicts remaining accesses*, enabling individual cost-benefit analysis for each pair of blocks.

### 3.3 Integration of RSM and MDM

MDM strives to maximize system performance, disregarding fairness. Within ProFess, we propose to steer MDM towards high fairness by using RSM as follows.

Upon an access to a block in M2, the program ID is known; let us denote it $c_{M2}$. Let us denote $c_{M1}$ the program ID of the program whose block is resident in M1 of the same swap group (the MC permanently stores $c_{M1}$ in the ST entry and updates it upon migration).

If $c_{M1}$ equals $c_{M2}$, the MC just uses MDM (Section 3.2.3) to decide whether to swap the blocks. Otherwise, we propose to guide decisions according to Table 7, where $SF_A(c_{M1})$ and $SF_A(c_{M2})$ are precomputed by (2) for $c_{M1}$ and $c_{M2}$, respectively, and $SF_B(c_{M1})$ and $SF_B(c_{M2})$ are precomputed by (3).

Case 1 in Table 7 indicates that $c_{M2}$ suffers from the competition for M1 more than $c_{M1}$ (both $SF_A$ and $SF_B$ support that). We propose an aggressive help strategy that forces swaps as if the program that needs help was running alone. Thus, in Case 1 the MC ignores the remaining accesses to the block of $c_{M1}$ and uses MDM as if M1 is vacant (Section 3.2.3). Recall that RSM is agnostic to the characteristics of M1 and M2, and so ProFess consults MDM regarding the benefit of the swap.

Case 2 in Table 7 represents the opposite case, where $c_{M1}$ suffers from the competition for M1 more than $c_{M2}$. Thus, according to our aggressive help strategy, the MC prohibits the swap, to protect the block of $c_{M1}$.

Case 3 in Table 7 is special, and its first condition indicates that $c_{M2}$ suffers more than $c_{M1}$ in terms of $SF_A$, while its second condition indicates the opposite in terms of $SF_B$. We find that for high fairness in this case, we must avoid *disproportionately* large $SF_B(c_{M1})$ by protecting the block of $c_{M1}$ as long as *the product* of $SF_A$ and $SF_B$ indicates that $c_{M1}$ suffers more than $c_{M2}$. This is reflected in the third condition of Case 3. In other words, the $SF_A \cdot SF_B$ products in the third condition of Case 3 allow us to assess whether $c_{M1}$ suffers more according to $SF_B$ than $c_{M2}$ suffers according to $SF_A$. The condition can be rewritten as $SF_B(c_{M1})/SF_B(c_{M2}) > SF_A(c_{M2})/SF_A(c_{M1})$, i.e., $SF_B(c_{M1})$ is greater than $SF_B(c_{M2})$ to a higher degree than $SF_A(c_{M2})$ is greater than $SF_A(c_{M1})$.

To exclude cases where $SF_A$ and $SF_B$ are too similar, we propose to use a small threshold in each condition in Table 7. For instance, using a threshold of ~3% (1/32, to simplify hardware), the first condition of Case 1 becomes $SF_A(c_{M1}) \cdot 1.03125 < SF_A(c_{M2})$. Since the third condition of Case 3 compares products of $SF_A$ and $SF_B$, we use a twice larger threshold there (i.e., $1/16 \approx 6\%$). If all three cases in Table 7 are false, the MC just uses MDM as per Section 3.2.3. Recall that migration decisions are not on the critical path.

**Table 7: Migration Decisions Guided by RSM**

*Block in M1 belongs to program $c_{M1}$; block in M2 to program $c_{M2}$*

| | |
|---|---|
| **Case 1** | $SF_A(c_{M1}) < SF_A(c_{M2})$ and $SF_B(c_{M1}) < SF_B(c_{M2})$<br>**Decision:** Consider M1 vacant and use MDM |
| **Case 2** | $SF_A(c_{M1}) > SF_A(c_{M2})$ and $SF_B(c_{M1}) > SF_B(c_{M2})$<br>**Decision:** Do not swap |
| **Case 3** | $SF_A(c_{M1}) < SF_A(c_{M2})$ and $SF_B(c_{M1}) > SF_B(c_{M2})$ and $SF_A(c_{M1}) \cdot SF_B(c_{M1}) > SF_A(c_{M2}) \cdot SF_B(c_{M2})$<br>**Decision:** Do not swap |
| **Default (all other cases):** use MDM | |

**Table 8: System Configuration**

| | |
|---|---|
| Num. cores / Core frequency | 4 / 3.2GHz |
| Core width / ROB size | 4 / 256 |
| Cache line size | 64B |
| Split L1 cache | 32-KB, 4-way, 2-cycle |
| Private L2 cache | 256-KB, 8-way, 8-cycle |
| Shared L3 cache | 8-MB, 16-way, 20-cycle |
| OS page size / Swap block size | 4KB / 2KB |
| ST entry size | 8B |
| STC | 64-KB, 8-way, 2-cycle |
| Num. memory channels | 2 |
| Channel frequency | 0.8GHz (1.6GHz DDR) |
| Channel width | 64b |
| M1 / M2 ranks per channel | 1 / 1 |
| Banks per rank | 16 |
| Rows per M1 / M2 bank | 1K / 8K |
| Columns per device / Device width | 1K / 4b |
| M1 / M2 row-buffer size | 8KB / 8KB |
| $t_{RCD\_M1}$ [26] / $t_{RCD\_M2}$ | 13.75ns / 137.50ns |
| $t_{WR\_M1}$ [26] / $t_{WR\_M2}$ | 15ns / 275ns |
| $CL$, $t_{RP}$ [26] | 13.75ns |

# 4. EXPERIMENTAL SETUP

## 4.1 System Configuration

We use a Pin-based [20] cycle-accurate x86 simulator [21] with a detailed DRAM main-memory simulator [22], which we modify to support NVM. Table 8 shows the system configuration. For multi-program evaluation, we simulate a quad-core system with two channels. To make the detailed-simulation time tractable (not longer than 3-4 days per workload), we scale the capacity of M1 to 256MB. Thus, the capacity of M2 is $8 \times 256$MB = 2GB (equivalent to eight times more rows per bank). For single-program evaluation, we simulate a single-core system with one channel and respectively scaled capacities of the L3 cache, STC, M1, and M2 (the total capacity in this system is 64MB M1 and 8×64MB = 512MB M2). We change cache sizes by changing the numbers of sets, and use CACTI [23] for latencies.

The row-to-column delay of M2, $t_{RCD\_M2}$, is ten times that of M1. NVMs typically have highly asymmetric latencies [16, 24], thus we assume a write-recovery latency $t_{WR\_M2} = 2 \times t_{RCD\_M2}$. The other timings of M1 and M2 are assumed identical, except that we appropriately adjust $t_{RAS}$ and $t_{RC}$ of M2, and that M2 has no refresh.

The MC employs the open-page policy and FRFCFS-Cap [25], limiting the number of consecutive row-buffer hits to four, which we modify to ignore row-buffer hits during swaps. We *accurately* model swaps, by issuing the read and write requests required to migrate data and by blocking the respective channel during a swap.

***PoM.*** We configure PoM using its default parameter values [5]. However, due to the characteristics of M1 and M2, for best performance we count each write request as eight accesses, and adjust the value of PoM's parameter $K$ using PoM's method [5], detailed below. One swap first reads a 2-KB block from M1 into swap buffer SB1 (Figure 1) and a 2-KB block from M2 into swap buffer SB2. Then it writes the blocks to M2 and M1, respectively. The read latencies ($t_{RP} + t_{RCD} + CL + 32 \times 4 \times 1.25ns$) partially overlap ($t_{RCD\_M2}$ overlaps with $t_{RCD\_M1}$, $CL$, and the majority of the 32 read bursts from M1). The write latency to M1 overlaps with $t_{WR\_M2}$ ($32 \times 4 \times 1.25ns + t_{WR\_M1} < t_{WR\_M2}$), given that

Table 9: Individual Programs [27]

| | MPKI | MB | | MPKI | MB |
|---|---|---|---|---|---|
| bwaves | 11 | 265 | mcf | 60 | 525 |
| GemsFDTD | 16 | 499 | milc | 18 | 547 |
| lbm | 32 | 402 | omnetpp | 19 | 138 |
| leslie3d | 15 | 76 | soplex | 29 | 241 |
| libquantum | 30 | 32 | zeusmp | 5 | 112 |

Table 10: Multiprogrammed Workloads

| | |
|---|---|
| w01 | mcf - libquantum - leslie3d - lbm |
| w02 | soplex - GemsFDTD - omnetpp - zeusmp |
| w03 | milc - bwaves - lbm - lbm |
| w04 | libquantum - bwaves - leslie3d - omnetpp |
| w05 | mcf - bwaves - zeusmp - GemsFDTD |
| w06 | soplex - libquantum - lbm - omnetpp |
| w07 | milc - GemsFDTD - bwaves - leslie3d |
| w08 | soplex - leslie3d - lbm - zeusmp |
| w09 | mcf - soplex - lbm - GemsFDTD |
| w10 | libquantum - leslie3d - omnetpp - zeusmp |
| w11 | soplex - bwaves - lbm - libquantum |
| w12 | milc - GemsFDTD - soplex - lbm |
| w13 | mcf - soplex - bwaves - zeusmp |
| w14 | GemsFDTD - soplex - omnetpp - libquantum |
| w15 | leslie3d - omnetpp - lbm - zeusmp |
| w16 | libquantum - libquantum - bwaves - zeusmp |
| w17 | mcf - mcf - omnetpp - leslie3d |
| w18 | mcf - milc - milc - GemsFDTD |
| w19 | milc - libquantum - omnetpp - leslie3d |

the 32 write bursts to M2 are issued before those to M1. Hence, the total analytic swap latency is 796.25ns. We observe that the average swap latency during our experiments is about 820ns (within 3% of the analytic one). The difference in 64-B read latencies of M2 and M1 is 123.75ns, which gives $K = \lceil 796.25/123.75 \rceil = 7$. Like the authors of PoM, we choose a slightly larger value, giving us $K = 8$. During execution the above latencies vary, but we observe that static $K = 8$ works well.

***MemPod.*** We find that in our system, MemPod performs best using 50-$\mu$s MEA intervals [7], 128 MEA counters, and counting each write request as one access. For optimistic evaluation, we ignore, just for MemPod, its overhead of ST updates upon swaps.

***ProFess.*** Each ProFess ST entry stores $4 \times 9 = 36$ address-translation bits, $2 \times 9 = 18$ QAC bits, and two program ID bits to keep track of the program whose block resides in the M1 location of the swap group, totaling 7B. We reserve another byte for future use (e.g., to store a wider program ID). The $min\_benefit$ value (Section 3.2.3) has the same meaning as PoM's parameter $K$, and we use $min\_benefit = K = 8$. Like for PoM, we count each write request as eight accesses. RSM employs 128 regions (such that four private regions are just $4/128 \approx 3\%$ of the total capacity) and a sampling-period duration ($M_{samp}$) of 128K requests (across the regions, per program). MDM uses 6-bit saturating STC Access Counters (AC, Figure 4). Thus, per STC entry, the counters require $9 \times 6 = 54$ bits, and the respective program IDs require $9 \times 2 = 18$ bits. For a 64-KB STC, that stores 8K 8-B ST entries, the storage overhead is $(54 + 18) \times 8$K bits = 72KB. Note that this is the *total* overhead per STC, that effectively allows us to track access counts per block, *for all blocks in M1 and M2*. The MDM observation and estimation phases are measured in updates of the MDM counters (Table 6), and the duration of each phase is 1K updates per program. The counters are reset at the beginning of each observation phase. During each estimation phase, the MC recomputes $exp\_cnt(q_I)$ using (5) every 100 updates per program. We choose such periodicity to obtain timely and reliable statistics.

## 4.2 Workloads

Like the recent works about hardware-managed hybrid memories [4, 5, 6, 7], we use SPEC CPU2006 [27] (Table 9), representing a large class of applications, and compose diverse multiprogrammed workloads (Table 10). The programs have various access patterns; for instance, mcf, omnetpp, and libquantum use irregular pointer-based data structures, and soplex has mixed regular and irregular accesses [28]. We use 500M-instruction simulation points [29] with the reference inputs. Table 9 shows the respective L3 Misses Per Kilo Instruction (MPKI) and footprints (in MegaBytes (MB)).

In each workload, we repeat programs that complete faster than the slowest one, ensuring competition for M1. Due to the repetitions, the aggregate number of simulated instructions varies: from 3B (w07) to ~8B (w05) and averaging 5B. Likewise, the total simulated execution time depends on the workload, ranging (for the baseline) from ~3B cycles (w15) to ~9B cycles (w09) and averaging ~6.5B cycles. The long simulations with multiple repetitions produce diverse overlaps of execution phases. In addition, due to the long simulations, M1 warms up relatively quickly: it gets utilized by 80% within the first 2% of execution time and by 90% within the first 3%, on average, across the workloads.

## 4.3 Figures of Merit

Weighted speedup is given by $\sum_i \left(sdn_i^{-1}\right)$ [15], where $sdn$ is given by (1) for each program $i$ in the workload. Unfairness is given by $\max_i (sdn_i)$ [13, 14]. We also estimate energy efficiency of the off-chip memory system as the number of requests served per second per watt, using power reported by the off-chip memory simulator [22]. Energy efficiency is more appropriate than execution energy, because the same workload might complete different amounts of work under different schemes (due to the different numbers of program repetitions).

# 5. EXPERIMENTAL RESULTS

We first present results for MDM in single-program experiments (Section 5.1), followed by sensitivity analysis (Section 5.2). Then, Sections 5.3 and 5.4 present results for MDM and ProFess (MDM + RSM) in multi-program experiments. We use the PoM migration algorithm as the baseline, since it represents the fairness problem, as we discuss in Section 2.4, and outperforms MemPod in our system, as we discuss in Section 2.5.

## 5.1 Single-Program Performance of MDM

Figure 5 shows the performance of MDM normalized to that of PoM for the programs from Table 9 in the single-core system. The results across the programs are

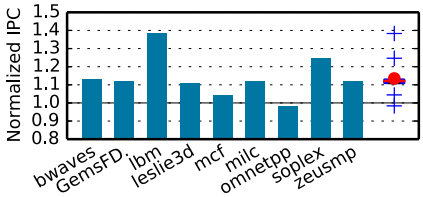


**Figure 5: Single-program IPC of MDM norm. to PoM**

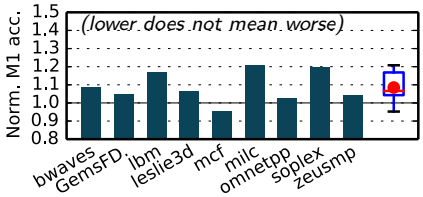


**Figure 6: Single-program M1 accesses of MDM norm. to PoM**

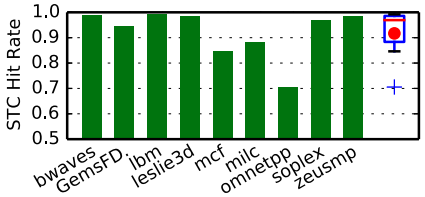


**Figure 7: Single-program STC hit rates under MDM**

summarized by a box plot [30], where the box shows the first and third quartiles, the whiskers show the data range, the "+" markers denote outliers, the red line denotes the median, and the red dot the geometric mean.

Figure 5 shows that MDM outperforms PoM by 14% (avg., up to 38% for lbm). We omit libquantum, since its footprint is just 32MB (Table 9) and fits entirely in M1, resulting in identical performance of MDM and PoM. However, we find that in an appropriately scaled system—with 4-MB M1 and 32-MB M2—MDM outperforms PoM for libquantum by 30%. The significant performance improvements confirm that MDM makes better migration decisions than PoM, thanks to its individual cost-benefit analysis. In addition, MDM reduces the average read-request latency by 18%.

Figure 6 shows the fractions of accesses served from M1 by MDM normalized to those of PoM. Higher fractions in Figure 6 correspond to higher performance in Figure 5, for all programs except mcf and omnetpp. To explain this, let us consider STC hit rates under MDM. Figure 7 shows that mcf has an STC hit rate of ~85%, reflecting its irregular accesses. Swaps of such blocks would hurt performance. We find that MDM identifies such blocks better than PoM and performs fewer swaps. This explains why MDM serves fewer accesses from M1 in Figure 6, but improves performance in Figure 5.

Next, Figure 7 shows that omnetpp has an STC hit rate of just 70%, reflecting its very irregular accesses. However, Figure 6 shows that MDM serves slightly more (by ~2.5%) accesses from M1. We find that the low STC hit rate reduces the accuracy of the MDM statistics, that misleads MDM, and it performs more swaps than PoM, which is unnecessary for omnetpp. This explains the insignificantly lower (by ~1.5%) performance of MDM for omnetpp in Figure 5.

Overall, performance improvements depend on multiple factors such as the amount of access irregularity, the ratio of a program's footprint to the total M1 capacity, and the amount of noise in the MDM statistics. Next, we present an analysis of sensitivity to selected factors.

## 5.2 Sensitivity Analysis of MDM

***Sensitivity to STC Size.*** MDM relies on the STC as a temporal filter for collecting statistics about data blocks. Premature ST-entry evictions from the STC due to conflicts or a lack of evictions due to no conflicts both add noise to the statistics, potentially hurting the accuracy of cost-benefit analysis. For instance, premature evictions may reduce $avg\_cnt(q_E)$ and the probabilities of $q_I$ transitioning to $q_E > q_I$, thereby reducing expected numbers of accesses $exp\_cnt(q_I)$ (Section 3.2.2). A lack of evictions would reduce the number of MDM counters' updates, and thus mislead MDM.

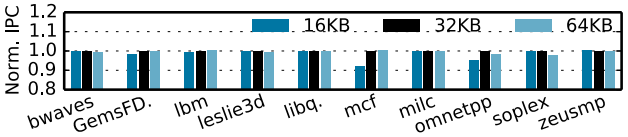


**Figure 8: Sensitivity to STC size**

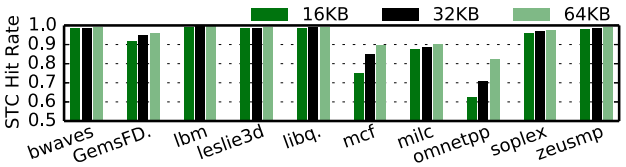


**Figure 9: STC hit rates vs. STC size**

Figure 8 shows IPCs of MDM with 16-KB and 64-KB STCs normalized to the 32-KB STC (the default STC size in the single-core system). Figure 8 shows that the programs are generally insensitive to STC size with a few exceptions. First, Figure 8 shows that mcf loses ~8% IPC when the STC is reduced to 16KB. Figure 9 shows the respective STC hit rates, where mcf suffers a drop from 85% down to 75%. The low STC hit rate increases the number of premature ST-entry evictions, adding noise to the MDM statistics and thus making it more difficult for MDM to make promotions that benefit performance. Similarly to mcf, omnetpp loses performance in Figure 8 when the STC size is reduced: it suffers a significant (~8%) STC hit-rate drop in Figure 9, which adds noise to its MDM statistics. However, with the reasonably sized STC of 32KB, MDM performs well.

Next, Figure 8 shows that a larger STC does not necessarily improve performance. For instance, omnetpp and soplex lose ~2% IPC when the STC size is increased to 64KB. The larger STC increases the STC hit rates in Figure 9, which reduces the number of ST-entry evictions, misleading MDM for omnetpp and soplex. For instance, we find that by forcing MDM counters' updates every 10M processor cycles after an ST-entry insertion to the 64-KB STC, we would increase the IPC of omnetpp by 1% and that of soplex by 3%. However, with the default STC size of 32KB, MDM performs well.

***Sensitivity to M2 Write Latency.*** Doubling the $t_{WR\_M2}$ latency increases the IPC improvement of MDM compared to PoM, making it 18% (avg., up to 61% for lbm). We find that MDM significantly reduces the frac-

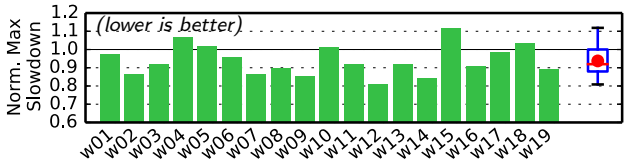


**Figure 10: Max slowdown of MDM vs. PoM**

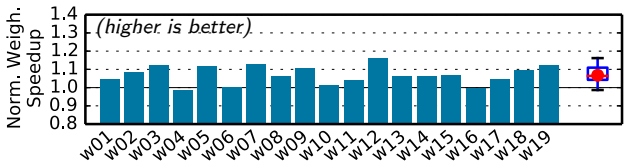


**Figure 11: Performance of MDM vs. PoM**

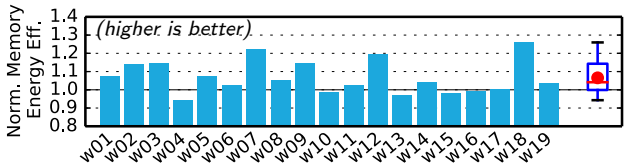


**Figure 12: Energy efficiency of MDM vs. PoM**

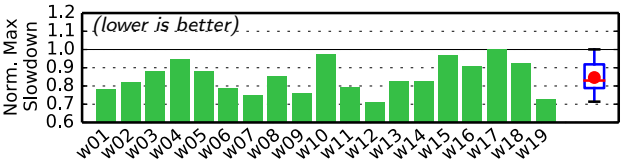


**Figure 13: Max slowdown of ProFess vs. PoM**

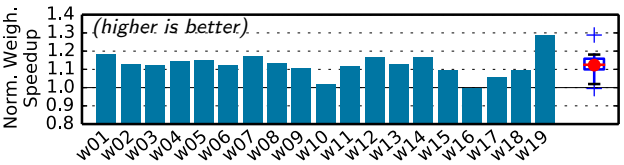


**Figure 14: Performance of ProFess vs. PoM**

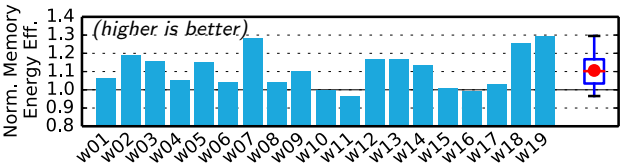


**Figure 15: Energy efficiency of ProFess vs. PoM**

tion of writes to M2 in programs where this fraction is above 1% and/or it significantly reduces the fraction of swaps (among all served requests). For the same reason, halving $t_{WR\_M2}$ reduces the IPC improvement of MDM, making it 12% (avg., up to 27% for lbm). For brevity, we do not show these plots.

***Sensitivity to M1:M2 Capacity Ratio.*** Changing the M1:M2 capacity ratio from 1:8 to 1:4 slightly reduces the IPC improvement of MDM vs. PoM, from 14% to 12% on average (excluding leslie3d, libquantum, and zeusmp, that fit entirely into the twice larger M1). However, changing the capacity ratio from 1:8 to 1:16 does not change the average IPC improvement, and it remains 14%. For brevity, we omit these plots.

## 5.3 Multi-Program Evaluation of MDM

Figure 10 shows the max slowdown of MDM normalized to that of PoM for the multiprogrammed workloads (Table 10) in the quad-core system. Figure 10 shows that MDM reduces the max slowdown—improves fairness—by 6% (avg., up to 19% for w12). This is solely because MDM speeds the programs in each workload. Figure 11 shows the respective system performance, where MDM outperforms PoM by 7% (avg., up to 16% for w12). In addition, MDM reduces the average read-request latency by 15%. Since performance is estimated as weighted speedup, it is possible to improve both performance and fairness. For instance, for w19 MDM improves fairness by 11% (Figure 10) and performance by 12% (Figure 11). However, for workloads w04, w05, w10, w15, and w18 MDM is less fair than PoM. This is not unexpected, since MDM ignores individual program slowdowns, just like PoM does.

Figure 12 shows that MDM improves memory-system energy efficiency compared to PoM by 7% (avg., up to 26% for w18). For w04 the efficiency is lower by 6%, which reflects its lower fairness in Figure 10 and no performance improvement in Figure 11. But for workloads like w03, a significant energy-efficiency improvement corresponds to significant fairness and performance improvements. Since MDM still suffers from fairness issues for some workloads, ProFess integrates it with RSM, and we present the respective results next.

## 5.4 Multi-Program Evaluation of ProFess

Figures 13 to 15 show respectively fairness, performance, and memory-system energy efficiency of ProFess normalized to PoM. Figure 13 shows that ProFess improves fairness by 15% (avg., up to 29% for w12), eliminating the fairness issues of MDM in Figure 10. Interestingly, for w17 ProFess does not find an opportunity to improve fairness. At the same time, Figure 14 shows that ProFess outperforms PoM by 12% (avg., up to 29% for w19), and Figure 15 shows that it improves energy efficiency by 11% (avg., up to 30% for w19).

In addition, ProFess reduces the average read-request latency by 9%, which is less than the average system-level performance improvement of 12%. This is so because ProFess significantly slows some cores to improve fairness, and the increased read latencies of those cores increase the average read latency. Another interesting observation is that ProFess reduces the fraction of swaps (among all served requests) by 24% (avg., up to 54% for w19). This is so because the proposed help policy prohibits some swaps, according to Table 7.

Next, we observe that there is no single relationship between fairness, performance, and energy efficiency. For instance, for w04 ProFess improves all three metrics, but for w11 the 20% fairness improvement in Figure 13 and the 11% performance improvement in Figure 14 correspond to a 3% lower energy efficiency in Figure 15.

Figure 16 shows slowdowns for the same workloads as in Figure 2. Unlike Figures 10 and 13, showing only the max slowdown per workload, Figure 16 details indi-

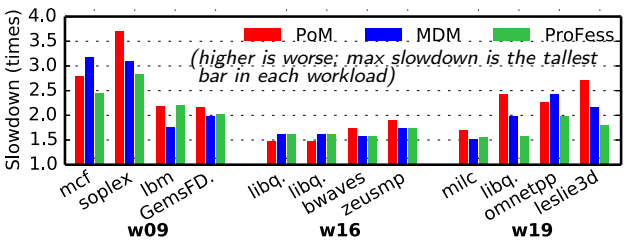


**Figure 16: Slowdowns under evaluated schemes**

vidual program slowdowns. Figure 16 shows that MDM can reduce the max slowdown solely by speeding programs (e.g., soplex in w09). ProFess further improves fairness, helping programs with high slowdowns by penalizing programs with lower slowdowns (e.g., in w09 ProFess slows lbm and GemsFDTD to speed mcf and soplex). Workload w16 is special, since ProFess finds no opportunity to improve fairness beyond that of MDM, achieved solely by maximizing performance. Figure 16 also illustrates that in a workload there is no equivalence between one program's performance increase and another program's performance decrease. For instance, in w19 ProFess increases the slowdown of milc by only ~2% compared to MDM, but at the same time reduces the slowdown of omnetpp by 22%. Thus, a lower max slowdown can correspond to higher performance.

Overall, the experimental results confirm our intuition that, in hybrid memories, fairness can be increased by allocating more M1 to a program that suffers the most from the competition for M1. We have also confirmed that changing the M1:M2 capacity ratio from 1:8 to 1:4 and from 1:8 to 1:16 affects the fairness and performance improvements in an expected way: increasing the ratio to 1:4 reduces the competition for M1 and therefore reduces the average improvements, while decreasing the ratio to 1:16 has the opposite effect, increasing the average improvements. Regardless of the M1:M2 capacity ratio, ProFess significantly improves both fairness and performance compared to PoM.

## 6. RELATED WORK

***Shared-Resource Management.*** Utility/fairness monitors guiding shared-resource allocation have been extensively studied in the contexts of shared caches [31, 32, 33, 34, 35, 36, 37, 38] and conventional main memories [39]. This paper considers a flat migrating hybrid memory, where the shared resource of interest—M1—is not a cache, and for practical reasons address translations (along with per-block metadata required for fair management) are stored in M1. Therefore, this paper considers a different memory organization, under different assumptions, which makes the techniques of the prior work not applicable.

***Slowdown Monitoring.*** Subramanian et al. [14, 40] use *request service rates* to proxy individual program slowdowns caused by memory-channel contention in conventional memories. Unlike the prior work, we address unfairness in *hybrid* memories, and propose a hardware technique (with OS support to maintain private and shared regions) to indicate which program suffers *the most* from *the competition for M1*, leveraging the intuition that, in hybrid memories with a large speed gap between M1 and M2, the competition for M1 is the major performance factor. To the best of our knowledge, this is the first proposal of slowdown monitoring in flat migrating hybrid memories. The proposed RSM can be integrated with other migration algorithms instead of MDM, since it merely guides migration decisions. RSM can also be used in systems with high-bandwidth DRAM as M1 and conventional DRAM as M2. Despite that the latencies of such M1 and M2 can be similar, the much higher bandwidth of M1 motivates migrations from M2 to M1 [4, 5, 6, 7]. Thus, co-running programs compete for M1, and the proposed RSM can identify a program suffering from the competition the most. Recall that RSM is agnostic to the characteristics of M1 and M2 and relies solely on the numbers of served requests and swaps.

***Migration Algorithms.*** Swap decisions have been conventionally based on the number of *already served* accesses, degrading individual cost-benefit analysis to a heuristic [4, 5, 6, 7], and Section 2.5 discusses the resulting performance problem. Unlike the prior work, we propose a conceptually new approach, where migration decisions are based on *statistically* predicted numbers of *remaining* accesses. This enables individual cost-benefit analysis for each pair of blocks, and makes the probability of promotions that *clearly benefit performance* the highest at a *first* access to a block in M2, i.e., when the number of *remaining* accesses to the block is the *greatest*. To the best of our knowledge, this is the first proposal of a probabilistic migration algorithm. The proposed MDM can be employed in hybrid memories with different M1-M2 address mappings, too. Recall that address-mapping relaxations are orthogonal to migration algorithms (Section 2.3). Thus if, for a given block in M2, more than one block in M1 gets chosen as a candidate for a swap (the choice is directed by possible address mappings, independently of MDM), MDM would just perform individual cost-benefit analysis for each pair of the blocks, and choose the most beneficial pair.

## 7. CONCLUSION

Flat migrating hybrid memory organizations can realize cost-effective main memories. Fair and at the same time high-performance management of such memories is an important challenge. This paper presents ProFess, a framework comprising: i) a relative-slowdown monitor and ii) a probabilistic migration-decision mechanism. This paper shows that ProFess improves fairness by 15% (avg.; up to 29%), compared to the state-of-the-art, while outperforming it by 12% (avg.; up to 29%).

## 8. ACKNOWLEDGMENTS

We thank Christos Dimitrakakis and the anonymous reviewers for their helpful comments. This work was supported by a grant from the European Research Council (ERC) under the MECCA project (contract 340328). We also acknowledge the support of HiPEAC.

# 9. REFERENCES


[1] D. Eggleston, "3D XP: What the hell?!!," in *Proc. Flash Memory Summit*, Aug. 2015.

[2] L. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proc. Int. Conf. on Supercomputing*, pp. 85–95, May 2011.

[3] H. Yoon, J. Meza, R. Ausavarungnirun, R. Harding, and O. Mutlu, "Row buffer locality aware caching policies for hybrid memories," in *Proc. Int. Conf. on Computer Design*, pp. 337–344, Sept. 2012.

[4] C. Chou, A. Jaleel, and M. K. Qureshi, "CAMEO: A two-level memory organization with capacity of main memory and flexibility of hardware-managed cache," in *Proc. Int. Symp. on Microarchitecture*, pp. 1–12, Dec. 2014.

[5] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent hardware management of stacked DRAM as part of memory," in *Proc. Int. Symp. on Microarchitecture*, pp. 13–24, Dec. 2014.

[6] J. H. Ryoo, M. R. Meswani, A. Prodromou, and L. K. John, "SILC-FM: Subblocked interleaved cache-like flat memory organization," in *Proc. Int. Symp. on High Performance Computer Architecture*, pp. 349–360, Feb. 2017.

[7] A. Prodromou, M. Meswani, N. Jayasena, G. Loh, and D. M. Tullsen, "MemPod: A clustered architecture for efficient and scalable migration in flat address space multi-level memories," in *Proc. Int. Symp. on High Performance Computer Architecture*, pp. 433–444, Feb. 2017.

[8] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," in *Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 631–644, Apr. 2017.

[9] M. Ekman and P. Stenstrom, "A cost-effective main memory organization for future servers," in *Proc. Int. Parallel and Distributed Processing Symp.*, pp. 1–10, Apr. 2005.

[10] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proc. Int. Symp. on Computer Architecture*, pp. 24–33, June 2009.

[11] A. Badam and V. Pai, "SSDAlloc: Hybrid SSD/RAM memory management made easy," in *Proc. Symp. on Networked Systems Design and Implementation*, pp. 1–14, Mar. 2011.

[12] The Next Platform, "Intel lets slip Broadwell, Skylake Xeon chip specs." `www.nextplatform.com`. Accessed: July 2017.

[13] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das, "Application-aware prioritization mechanisms for on-chip networks," in *Proc. Int. Symp. on Microarchitecture*, pp. 280–291, Dec. 2009.

[14] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing performance predictability and improving fairness in shared main memory systems," in *Proc. Int. Symp. on High Performance Computer Architecture*, pp. 639–650, Feb. 2013.

[15] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, pp. 42–53, May 2008.

[16] M. K. Qureshi, S. Gurumurthi, and B. Rajendran, *Phase Change Memory: From Devices to Systems*, vol. 6 of *Synthesis Lectures on Computer Architecture*. Nov. 2011.

[17] Diablo Technologies Inc., "Memory1." `www.diablo-technologies.com/memory1`. Accessed: July 2017.

[18] ScaleMP, Inc., "vSMP Foundation Flash Expansion." `www.scalemp.com/products/flx`. Accessed: July 2017.

[19] R. M. Karp, S. Shenker, and C. H. Papadimitriou, "A simple algorithm for finding frequent elements in streams and bags," *ACM Trans. Database Syst.*, vol. 28, pp. 51–55, Mar. 2003.

[20] Intel Corp., "Pin – a dynamic binary instrumentation tool." Pin 2.12 User Guide. `software.intel.com`, Apr. 2013.

[21] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, T. Pho, H. Kim, and R. Hadidi, "MacSim: A CPU-GPU heterogeneous simulation framework." User Guide. Georgia Institute of Technology, Sept. 2015.

[22] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A cycle accurate memory system simulator," *IEEE Computer Architecture Letters*, vol. 10, pp. 16–19, Jan. 2011.

[23] HP Labs, "CACTI: An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model." `http://www.hpl.hp.com/research/cacti`. Accessed: July 2017.

[24] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, pp. 994–1007, July 2012.

[25] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *Proc. Int. Symp. on Microarchitecture*, pp. 146–160, Dec. 2007.

[26] Micron Technology, Inc., "Micron® 4Gb: x4, x8, x16 DDR4 SDRAM features." Datasheet, `www.micron.com`, 2014.

[27] SPEC, "SPEC CPU2006." `www.spec.org/cpu2006`.

[28] A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proc. Int. Symp. on Microarchitecture*, pp. 247–259, Dec. 2013.

[29] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "SimPoint 3.0: Faster and more flexible program analysis," *J. of Instruction-Level Parallelism*, vol. 7, pp. 1–28, Sept. 2005.

[30] J. W. Tukey, *Exploratory Data Analysis*. 1977.

[31] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. Int. Symp. on Microarchitecture*, pp. 423–432, Dec. 2006.

[32] R. Iyer, "CQoS: A framework for enabling QoS in shared caches of CMP platforms," in *Proc. Int. Conf. on Supercomputing*, pp. 257–266, June 2004.

[33] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 111–122, Sept. 2004.

[34] J. Chang and G. S. Sohi, "Cooperative cache partitioning for chip multiprocessors," in *Proc. Int. Conf. on Supercomputing*, pp. 242–252, June 2007.

[35] D. Kaseridis, J. Stuecheli, and L. K. John, "Bank-aware dynamic cache partitioning for multicore architectures," in *Proc. Int. Conf. on Parallel Processing*, pp. 18–25, Sept. 2009.

[36] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and efficient fine-grain cache partitioning," in *Proc. Int. Symp. on Computer Architecture*, pp. 57–68, June 2011.

[37] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic shared cache management (PriSM)," in *Proc. Int. Symp. on Computer Architecture*, pp. 428–439, June 2012.

[38] H. Kasture and D. Sanchez, "Ubik: Efficient cache sharing with strict QoS for latency-critical workloads," in *Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 729–742, Mar. 2014.

[39] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, "Dynamic tracking of page miss ratio curve for memory management," in *Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Syst.*, pp. 177–188, Oct. 2004.

[40] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory," in *Proc. Int. Symp. on Microarchitecture*, pp. 62–75, Dec. 2015.