Soft2LM: Application Guided Heterogeneous Memory Management

Michael Giardino
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia
Email: giardino@gatech.edu

Kshitij Doshi
Intel Corporation
Chandler, AZ
Email: kshitij.a.doshi@intel.com

Bonnie Ferri
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia
Email: bonnie.ferri@ece.gatech.edu

***Abstract*—This paper introduces a software policy for memory management in heterogeneous memory systems in order to improve the trade-offs between performance and power consumption, while attempting to make the best use of different characteristics of the underlying memory technologies. In this policy, the operating system and the application co-schedule page management in order to make informed decisions about page allocation and migration. Software-Controlled 2-Level Memory (Soft2LM) is a hardware-agnostic approach for efficient usage of heterogeneous memory that allows for region-based allocations, migrations, and application advice. We include analysis of the access characteristics of the PARSEC 3.0 suite of benchmarks, both as motivation for software-guided intelligent page placement as well as for understanding where application advice can make a difference. Our evaluation running PARSEC on our Soft2LM Linux kernel shows an average 4.7x improvement over comparable RAMDISK-based swapping. Even when migrating 100 and 1000 pages per second between regions, Soft2LM performs comparably to a stock system with twice the available DRAM, showing a 1.8% and 0.7% improvement, respectively.**

I. Introduction

Heterogeneous memories, byte addressable storage elements with non-uniform characteristics populating a shared address space, are on the horizon. Among these, *storage-class memories* (SCM) [1] are being discussed as exciting candidates for augmenting DRAM in servers due to their characteristics of large-capacity, persistence, and accessibility at lower latency and higher bandwidth in relation to high performance block storage devices.

Since they offer both persistence and high capacity, and allow much faster direct access to content in comparison to that allowed by disks, it is very attractive to use them to eliminate I/O stalls from programs. However, due to technology differences between SCM and DDR3/4, they are expected to exhibit measurably higher and non-uniform latency, and variable, lesser bandwidth, in comparison to conventional volatile memory. The ideal combination from software standpoint is to hold frequently accessed data in DRAM, and use the SCM memory that shares a program's address space, to support the long tailed references to the remainder (which would have otherwise missed the DRAM page cache and generated disk requests). Intelligent placement of rarely touched data in SCM may also be used to save power, even if a program has a sufficiently small total footprint that available DRAM can accommodate.

To benefit from this opportunity to run programs without I/O stalls and with reduced power consumption, while simultaneously achieving high performance, it is critical to ensure that a dataset is hosted in DRAM while it is popular and offloaded to SCM as its popularity declines, and then eventually moved out of the memory tier altogether. Processor caches achieve such dynamic placement directly in hardware–usually at cacheline granularities; operating systems, language runtimes, virtual machines, and application software need to perform a similar dynamic placement, except where hardware/firmware based mechanisms are available for managing DRAM as a memory-side cache in front of SCM. Even when hardware based "transparent" caching of SCM-based content in DRAM is available, it is generally difficult to tailor the caching policy to capture a warm subset of data whose dynamic footprint exceeds the capacity of processor caches.

Software based policies that attempt to maximize data residency in a performance tier may be divided into three classes: at one extreme, the operating system makes the decision of what to evict (i.e., similar to conventional paging), at another, it is the application software that explicitly directs the operating system in placement and displacement decisions (i.e., similar to the manner in which a database performs its record buffering). A hybrid policy might keep some control in the operating system so that applications are freed from the minutiae of managing the memory resources, but are able to work in an enlightened way together with the operating system to avert the large majority of page faults from untimely evictions of warm data. The contribution of this paper falls into the third category, where a number of nuances need to be addressed to streamline the process of reclaiming memory from one use and assigning it to another.

In particular, the paper deals with the problem of co-scheduling page management activities between application software and operating system software. One challenge is to ensure that data that is currently hosted in a DRAM page is moved over to an SCM page without requiring expensive synchronization over TLBs. Conventional "lazy TLB shoot-down" approaches are less effective since reassignment of

pageframes between SCM and DRAM needs to be performed in a timely manner. A consideration that applies in a discriminating reassignment is only the pages that are modified since they were given DRAM placement need to have their data migrated to SCM, where as for clean data, their SCM copy can be substituted at the same virtual address without such data migration. Another consideration that applies is the cost of data migration: it is not prudent to promote data from an SCM range into DRAM placement unless there is convincing evidence that the data will be used frequently enough to justify (amortize) the cost of copying. Applications may further make static choices: for instance, write-only or write-mostly ranges such as file system journaling pages or data logging pages benefit scarcely from DRAM placement; and data that is updated but is temporary (e.g., Java nurseries) may not benefit sufficiently from SCM placement; such pages need to be managed with input from the application. On the other hand, an application's behavior may not be sufficiently predictable, so that from one phase of program behavior to another, it is not prudent to associate either an SCM page or a DRAM page continuously with a given virtual address. Thus a collaborative policy in which applications make their intent as transparent as possible while a memory management system supports the intent but protects against continued deviation from it is desirable and this paper proposes such a scheme.

The paper proposes a page life cycle that draws upon current page life cycle in Linux 3.x version, but extends it for flexible application of decisions about when to migrate and when to flush TLBs globally. Because large capacity SCM [Intel 3DXP] is not yet available, we analyze memory access traces and statistics under a number of memory intensive applications to explore the options noted above.

- We present a hardware agnostic approach for efficient usage of heterogeneous memory. We term the approach "Software Controlled 2-level Memory" (Soft2LM). Soft2LM is built on a 3.x Linux kernel, and it allows for region-based allocations, migrations, and application advice.
- As part of Soft2LM, we built an extensible API that allows applications to describe their memory usage characteristics to the operating system, in page-granular units, through which the operating system receives and implements responsive page placement and data migration.
- In our evaluation, we execute the PARSEC suite using virtualization in order to have control over the hardware available to the OS and show that the overhead of comprehensive hybrid page management requires little overhead even when the memory system is heavily taxed. The implementation also significantly outperforms low-latency swapping to a RAMDISK, improving over a similarly sized RAMDISK by 4.7x.
- Using the memory footprint and access statistics collected under PARSEC, we establish that Soft2LM's cooperative memory management effectively blends the supplied usage guidance into operating system's memory management, and that this efficiently counteracts the oversubscription of DRAM by an application.

II. Background and Related Work

Owing to its high capacity, low latency, and persistence, SCM can be integrated into current machine organizations in a multiplicity of ways: for example, as memory, as storage, as a hardware or software managed cache for either, and various combinations thereof. A consensus is growing around what is known as a hybrid memory system [2]–[4]. In these systems, NVM is logically inserted into the memory hierarchy between DRAM and SSD, where it used as a transparent hardware-managed disk cache or as a shared-address space main memory, managed by the operating system's memory manager. This paper extends the DRAM-as-a-cache for SCM usage in two ways: it implements an efficient type of memory-to-memory paging in software, and in doing so, it also assigns to application software a steering role so that data is promoted or demoted between DRAM and SCM based on a combination of application input and application reference behaviors.

As with mixed SCM and DRAM arrangements, memory access latencies and bandwidths also vary in DRAM-only NUMA designs in which different latency and bandwidth characteristics apply for intra vs inter domain access [5]. In general, balanced memory population employed in most hardware systems creates equal capacity NUMA domains where as the capacity of an SCM tier is expected to be significantly greater than that of a DRAM tier. Inter-domain NUMA access latencies are expected to be insiginificant in comparison to inter-tier latencies in heterogenous memory. These asymmetries, combined with SCM tier's persistence and power characteristics encourage a more informed scheduling of memory on the basis of joint participation by applications and the runtimes that host them in heterogenous memory. Shin et al expand on existing NUMA architecture in the Linux kernel, assigning NVM and DRAM to different NUMA node IDs, allowing the operating system to use existing NUMA migration to move pages between NVM and DRAM [6]. Adjacent physical pages are allocated into groups and to determine hotness of data, they use unused bits in the page table entry to store a weighted history of the pages' dirty bit. Their results show that even using separate NUMA domains, the overhead of migration in total execution time is only 1.14%, and they get a 19-36% decrease in energy consumption. We bypassed separate NUMA nodes, further reducing the codepath of different regions of memory. This also leaves NUMA available to build onto a heterogeneous system in the conventional way.

Lee describes a technique for using hybrid memory management that uses a hypervisor to scan and extract page access histories for tasks in guest virtual machines [7]. Based on these, the hypervisor can perform intra-VM and inter-VM allocation of capacity in on-chip stacked DRAM modules, so that relatively expensive off-chip DRAM accesses can be minimized for frequently referenced pages. Our technique is similar but does not require a hypervisor as an intermediary,

and therefore generalizes to single as well as multitenant execution. Hardware based techniques for managing stacked DRAM modules are described by Sim [8] and Chou [9]. Sim proposes a Part-of-Memory (PoM) architecture in which a page activity tracker guides hardware in remapping hot data to on-chip tier [8] , while Chou proposes a CAche-like MEmory Organization (CAMEO) for migrating referenced cachelines into on-chip tier while furnishing the total capacity of on-chip and off-chip DRAM to software [9] . A common characteristic of both approaches is that the latency and capacity of the stacked DRAM are comparable to those of last level caches, making the hardware approaches more fitting. Our software-based approach makes it possible for applications to benefit from both the larger capacity of the near tier and persistence of the far tier as well as the ability to draw upon a long history of page access patterns to make informed decisions about allocation and placement.

Intelligent page placement is also explored in the context of hybrid systems comprising CPUs and GPUs by Agarwal [10] and Li [11]. Agarwal developed bandwidth-aware page placement driven by both compiler extracted insights and explicit hints from software is used to show 35% improvements in GPU performance [10]. Their experiments show a marked improvement, but because hardware amenible to their algorithm was not available, they were forced to conduct their experiments in the simulator. Results from Li show that in comparison to a hardware managed approach, static assignment of hot data to on-chip DRAM doubles performance and cuts power consumption in half [11].

For hybrid memory systems comprising off-chip DRAM and off-chip SCM modules, a hardware memory controller is proposed by Ramos [12]; the controller monitors access patterns and remaps pages, while maintaining its own address translation table to keep such data movement transparent to application and operating system. Using simulation for such hybrid systems, Meza et al show power consumption and performance improvements under a hardware-based but software-assisted blended memory management approach – with significant gains resulting from displacing a hard disk drive with SCM, and further gains arising from eliminating software overhead of file system calls [13] . This is in line with our findings–that in lieu of using SCM as a faster paging device, it is a better choice to eliminate the I/O and serialization overheads by using SCM as extended memory into which colder virtual addresses are mapped. Using trace driven simulation, Seok et al conclude that it is imperative to reduce write accesses and energy consumption by taking into account the non-uniform latency and endurance of SCM [14]. Others have examined low-level hardware modifications to improve performance, reduce power consumption, and improve device lifetime: Yoon explored row-buffer aware caching policies [15], Qureshi optimized DRAM cache architecture for latency, even at the expense of hit-rate, by both reducing the associativity and streaming data and tag in a single data burst, and introduced a memory access predictor [16] while Meza added a small cache for recently used metadata in DRAM [17].

The approach proposed in this paper is driven from a similar perspective but different vision: that due to their non-uniform power, latency, and bandwidth characteristics, SCM accesses need to be reduced to a minimum and that such reduction can be pursued sooner with software approaches. while minimal hardware extensions (e.g., in data access instrumentation) that can assist software are also identified sooner as a side effect.

In order to enable software-guidance of memory access patterns, `madvise()` is a system call that allows programs to advise the kernel as to the anticipated access characteristics of a given range of virtual memory [18]. This advice can be used by the kernel when prefetching data and freeing pages. For example, a program can specify `MADV_SEQUENTIAL`, which informs the kernel that it can aggressively read-ahead and discard the used pages when they are finished. When dealing with heterogeneous memories, the decisions the kernel makes have many potential benefits but it becomes a more complex decision. Instead of having a single piece of advice, we implement a set of flags that can, like GFP flags, be combined to describe multiple characteristics of a page.

Jantz et al developed a framework to "color" pages in order to provide application guidance to the operating system for physical page placement [19]. Using *trays* to organize and place data on specific physical DRAM pages based upon application guidance showed 55% power improvements on a synthetic benchmark. Our kernel API is similar but provides additional options for applications to identify data in heterogeneous memory systems.

`jemalloc` is a scalable concurrent `malloc` replacement which uses isolated regions of memory called *arenas* to reduce lock contention in CMP systems. It also uses `madvise` to release pages back to the system. Building upon `jemalloc`, memkind extends the well-known application allocation API (`malloc`, `free`, etc.) with the kind of memory requested [20]. By using the arenas of `jemalloc`, memkind is able to allocate memory of the desired type in a specific region of memory corresponding to the requested kind.

III. System Overview

We are proposing two additions to page management: the ability to guide page placement and monitor page usage, and a mechanism for transparent page movement.

A. Application Guidance

Application guidance of page placement requires an API that allows for anticipated use of memory allocations to be relayed to the operating system. This differs from `madvise` in that `madvise` is called after the mapping has been made. Our modifications pass information to the operating system at allocation time. We propose the needed information be passed to the kernel as additional flags in the existing `mmap()` call. Initially, the modifications are requests for specific regions of memory, `MAP_PREF_NVM` and `MAP_PREF_DRAM`, which map to the modifications in page allocation flags outlined in table I and discussed below. There is a lot of potential for

TABLE I
SOFTWARE ADVISE FLAGS FOR PAGE ALLOCATION USING `MMAP()`

Basic, fail only with ENOMEM	
`MAP_PREF_NVM`	prefer NVM but accept DRAM
`MAP_PREF_DRAM`	prefer DRAM but accept NVM
Basic, fail if no required memory is available	
`MAP_REQ_NVM`	require NVM
`MAP_REQ_DRAM`	require DRAM
Usage based flags	
`MAP_READ_ONLY`	Write triggers protection fault and reflagging
`MAP_READ_MOSTLY`	Latency consideration in allocation, try to keep clean (writeback)
`MAP_WRITE_MOSTLY`	Latency insensitive, placement in far, persistent, and low-energy memory
`MAP_SCRATCHPAD`	For temporary data, latency and physical endurance primary concerns
`MAP_LATENCY_SENSITIVE`	Data without specific read and write patterns, but expected to be latency sensitive, similar to `reg` in C
`MAP_REQ_PERSIST`	Require placement in persistent memory or force write-through
Anticpated Use Patterns	
`MAP_HOT`	Data that is part of a critical set, allocate in near memory and attempt to keep close to CPU
`MAP_COLD`	Data that is expected to be used infrequently, allocate in far memory

use-based flags that can then be interpreted by the operating system based upon it's available physical memory.

If the application only prefers a specific type of memory (`MAP_PREF_*`), then the allocation will only fail if no memory is available when a page is actually allocated. Because `mmap()` uses lazy allocation and only brings in data when there is a page fault, the available memory may change during runtime, using preferences (as opposed to requirements) will prevent future allocations from failing.

If an allocation is required to be in a specific region, then we can specify that as well. If an application requires persistence of data (e.g. critical logs) `MAP_REQ_NVM` will ensure that data is written to non-volatile memory. If the application needs low-latency access or is doing heavy but temporary writing (such as a scratchpad), `MAP_REQ_DRAM` can be selected. When used in conjunction with the existing flag `MAP_POPULATE`, the page table will be prepopulated and the file will be read-ahead. This should ensure that, at allocation time, `mmap()` either gets the memory in the required region or fails immediately. If `MAP_POPULATE` is not used, there is the potential for a failure when a page fault occurs later.

Ideally, we would like for an application to be able to simply specify the expected use pattern for a given region of memory and have the operating system make allocations based upon the physical memory available, potentially leveraging many types of heterogeneous memories. Common use patterns could

Fig. 1. A sample epoch showing the large computational period and the smaller migration-related periods.

be defined by the flags in table I. Read only pages could be moved into DRAM as needed, but then moved down to NVM as they cool, and eventually being discarded without writeback when space is needed. Read-mostly pages, anonymous or file-backed, when located in DRAM could be occasionally synced with NVM or disk when dirtied, but otherwise kept close to the CPU for low latency accesses. Write-mostly regions of memory can be reserved for those where data is written consistently but without the need for low-latency reads, and thus can be allocated in far memory.

Scratchpad regions of memory are those which store temporary data used for ongoing calculations or processing. These data may experience a lot of writing and reading and thus should remain low-latency for both reading and writing, and in physical memory with high endurance. Latency sensitive memory regions are those which are known to be accessed frequently but may not have the high write access patterns, thus endurance is not a factor in selecting the underlying pages. If data needs to be kept persistent, the require persistence flag can force all allocations into non-volatile regions of memory, or in non-heterogeneous memory force write-through.

In addition to specific use patterns, the actual access frequency of the data can be specified as hot or cold. Hot data is given a preference for near memory, however the operating system may move data around as it sees fit. Cold memory is allocated in far memory and will not be moved to near memory without good reason (e.g. `madvise` update or operating system monitoring).

B. Migration Mechanism

The migration mechanism is built upon the Linux's memory management. It contains three primary changes to page management: split physical memory, epoch-based page migration, and region-aware memory management.

The physical address space is split into two contiguous regions, each one having its own data structures including per-order free lists, per-cpu active/inactive LRU lists, and page caches. *Epoch-based migrations* are used to amortize the cost of moving pages and remapping, as well as reduce thrashing. An epoch, shown in Figure 1, allows the memory management system to work in stages, collecting performance and power data during the execution of applications, using this data to make decisions about page movement, selecting pages to move, copying the data, and then committing the movement at the end of the epoch. Performance data is currently collected by the memory subsystem and performance counters, however it is expandable to receive data from other sources such as

hardware sensors, daemons, as well as an application's own knowledge of its performance.

The decisions about data movement are similarly flexible. The initial implementation uses DRAM capacity thresholds to trigger a migration of data from near memory to far memory. The migration mechanism looks on the inactive lists for pages that can be moved to NVM. It is important to note that since the second level memory is byte-addressable, while there may be a performance penalty associated with the additional latency of second level, there is not the danger of swapping or having to re-read a page from disk. The page, once the migration is complete, will be accessible by applications without suffering a page fault.

If the stall cycles of a CPU begin to increase indicating a program becoming increasingly memory bound, the migration mechanism can work in reverse, taking pages from the second-level memory's active list, and transparently migrating them to DRAM. Future work will develop more sophisticated methods of identifying candidates for migration.

Whenever a shared page is remapped, any CPUs which have the page mapped must remove from the TLB the cached entry. In order to ensure that no stale mappings are used, the remapping CPU issues a *TLB shootdown* to remove the stale entry from all processors. This synchonization requires an inter-processor interrupt (IPI), an expensive operation that stalls the issuing CPU until all CPUs that receive the IPI acknowledge that they have removed the TLB entry. Currently, the actual act of migrating the data occurs right before the batched remapping, however the system could be expanded to perform data migrations in the background, staging the moved pages for a batched remapping at a later point.

C. Page Selection

The migration begins upon the expiration of a high resolution timer whose duration can be set via a `/proc` file. This timer interrupt places the migration task on a workqueue and returns. When the workqueue is executed, the selection of pages begins. The kernel checks the threshold of DRAM as well as the maximum number of pages to be moved, both set via `/proc` filesystem entries, to determine how many pages need to be moved. Much like `vmscan` scans LRU vectors `lruvec` to age and free inactive pages, the migration mechanism scans these vectors looking for migration candidates. These LRU vectors are divided into four LRU lists: inactive file-backed pages, inactive anonymous pages, active file-backed pages, and active anonymous pages. In this order, the page vectors are scanned, first looking for unmapped pages that are clean and can be migrated asynchronously. Inactive, clean, unmapped file-backed pages may be discarded without penalty. However at this time, they are migrated under the assumption that second-level memory is large enough to accommodate most if not all of the total memory footprint. In addition, due to the persistent nature of NVM not requiring energy to refresh as in DRAM, once data is written back to NVM, there is no cost in keeping it there indefinitely until there are capacity constraints.

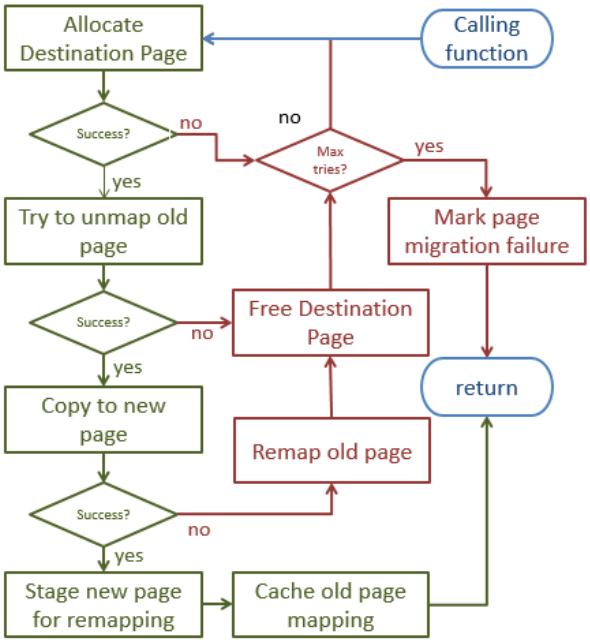

Fig. 2. Detailed flowchart of migration.

If moving unmapped pages is unsuccessful in getting the needed number of pages, the list is scanned for mapped, clean pages. Finally, the list is scanned for dirty pages. It is important to keep in mind that we are not writing back dirty pages when we migrate; there is no reason to worry about the page state when moving them to another active region of main memory. We are however using the dirty state of the page as additional information on the activity of the page. Since `vmscan` regularly passes through main memory, looking for pages to free, we can use the fact that it is dirty to assume that the page may be recently used. If the threshold can be met by simply moving cold/inactive pages to SCM but keeping them mapped, the pages are collected and staged to be moved. If there are not enough pages in the inactive list, pages are collected from the active lists. Once the candidate pages are selected, each page is copied to its new location. When the TLB flush is acknowledged by all nodes, the epoch timer is reset and the kernel cedes.

D. Migration Process

The detailed process of page migration is shown as Figure 2. It follows closely the migration used in Linux already by compaction, memory hotplug, and memory failure paths, however there are significant differences in implementation, due to different failure modes, allocation sources, and page selection. Once we have found candidate pages to migrate from a region, the calling function attempts to allocate a page in the needed region using a pair of new page flags: `GFP_NVM` and `GFP_MIGRATE`. If we are unsuccessful in allocating a page, going all the way from page caches, down to the buddy system, we unwind and try again. Once we obtain a page, we attempt to unmap the old page. If this is unsuccessful, the

destination page is freed and we try to allocate a new page. Once the page is unmapped, all the data and meta data are copied to the new page. While rare, it is not impossible for the data to not copy correctly, so if the data copy fails the page is remapped, and the destination page freed. If the copy is successful, the page is remapped, the old page mapping is cached, and the migration is complete. The caching of old page mappings is an important addition that will allow for *page aliasing*. Since a page, once migrated, is not immediately freed, but rather cycled through various states of cleanliness and activity, there is a good chance that at least for a while, the page may exist in two places at once, that is DRAM and NVM. This provides an interesting opportunity for an even more lightweight case of migration and an extremely powerful use of Soft2LM. Take for example a page that is NVM, but has been accessed often and thus should be moved to DRAM. If we have the old page mapping cached and the NVM version is clean, instead of writing the page back from DRAM, costing bandwidth and write power, we can simply use the cached mapping to stitch the page back into the page table, and move the DRAM version of the page to an inactive list for future freeing.

IV. Experimental Evaluation

There are three facets of evaluation to consider. First, given the added features of enumerated regions of memory and the modifications of memory management allocation and freeing mechanisms, we measure the introduced overhead by comparing the performance of a split-memory tiered memory manager with an unmodified kernel. Second, in order to show the benefits of a shared address space over block-based storage, we compare our tiered-memory manager with swap placed on a DRAM-based ramdisk. The choice of a ramdisk eliminates any device latency penalty that swapping would normally introduce, comparing only the costs of the mechanisms. Third, in order to evaluate the overhead of moving active memory pages during benchmark execution, we migrate 100 and 1000 pages per second, and compare the results to swapping.

Because our focus is on applications with large data needs, we chose the PARSEC benchmarking suite that contains 13 varied memory intensive workloads. [21]. Experiments were carried out in a KVM/qemu-based virtual machine running on an Intel Haswell i7-4770. Virtual machines with 4 Haswell cores (2 physical, 4 logical) and 6 GB of DRAM were created running Debian 8 and a modified version of the Linux kernel v3.14.39. Since there is no consumer-grade SCM available, our experimental evaluation was run using two regions of DRAM, and for the comparison with Linux's paging, a RAMDISK used as a swap partition.

A. PARSEC

The PARSEC v3.0 suite was selected for its diverse set of memory intensive workloads and its use in current research. All benchmarks were run with 4 threads in an attempt to reduce the potential for CPU bottlenecks, which would reduce the stress on the memory system. The `native` size dataset was used for all runs. PARSEC measures the execution time of the benchmarks by using the Unix `time` command, giving us three time values for each run. The **real** time is the actual wall time elapsed during the execution of the benchmark, the **user** time is the time the process was executing on the kernel, and the **system** time is time spent in the kernel.

TABLE II
Memory Characteristics of simsmall and native PARSEC Benchmarks

	native (MB)		simsmall (unique pages)			
Benchmark	RSS	VSZ	unique	read	write	inst (M)
blackscholes	611	652	190	188	95	106
bodytrack	31	380	2096	1884	1868	301
canneal	938	1239	10630	10628	10430	607
dedup	1681	2621	6778	6777	4107	764
facesim	302	552	80326	65669	80073	11848
ferret	102	1330	2409	2394	1574	519
fluidanimate	597	642	19039	10176	18851	440
freqmine	705	941	12271	12231	12038	921
raytrace	1124	1444	42920	40261	42602	9519
streamcluster	106	223	419	417	232	420
swaptions	4	229	428	410	192	697
vips	41	361	1997	1588	1184	966
x264	181	310	2546	2440	2361	218

1) Memory Footprint of `native` PARSEC benchmarks: In order to determine the actual memory footprints of the PARSEC suite, during the runtime of all the benchmarks, the resident set size (RSS), virtual memory size (VSZ), and major and minor page faults were sampled every second and logged to a file. In the output of `ps`, RSS is the amount of an application's memory that is allocated and actually resident in RAM (i.e. non-swapped) while VSZ is the total size of the virtual memory including code, shared libraries, and swapped out portions of code. Table II shows the measured footprint while running the `native` size dataset. What is interesting about these measurements is how a significant portion of them do not have especially large datasets. For example, swaptions has only 4MB resident in RAM at any given time, while both bodytrack and vips have less than 48MB. Further work is being done to better profile these benchmarks to understand both their temporal access patterns and implementation of application guided allocations.

2) PIN-based traces of `simsmall` PARSEC benchmarks: In addition to running the full benchmarks on a full system, we examined the access properties of this benchmark suite. We used PIN [22] to capture memory traces of the PARSEC suite. Since the `simsmall` datasets generated nearly 700 GB of traces, the longest being over 10 B instructions, it wasn't possible to directly compare these data to the `native` datasets, however, we believe it gives some insight into the page access patterns, further motivating the need for intelligent memory management.

The right half of table II shows the unique pages recorded, and the number of pages read and written, as well as the number of memory instructions collected for the entire simsmall run. Here we can see the differentiation of written and read pages across all benchmarks. For example, in `canneal`,

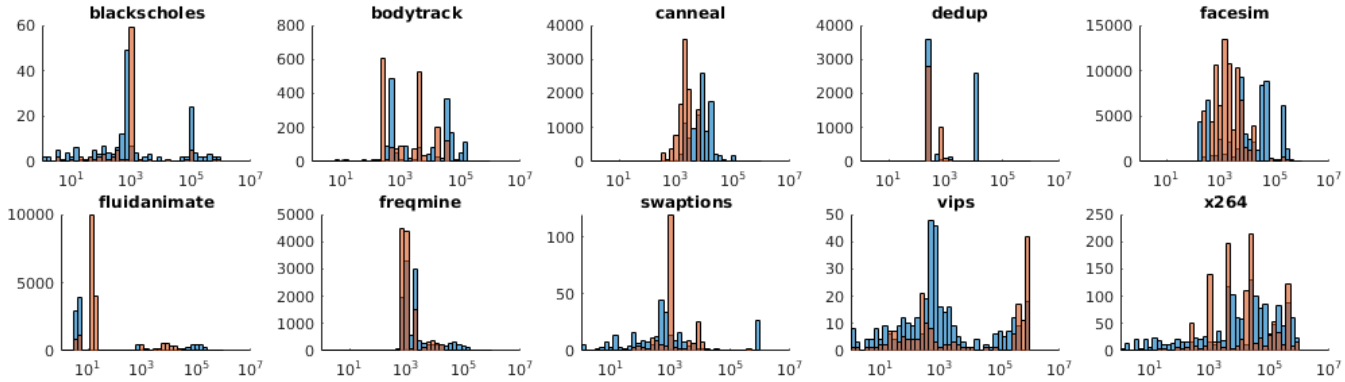

Fig. 3. This figure shows the page access histograms for the PARSEC 3.0 suite of benchmarks running the `simsmall` datasets. Blue indicates page accesses for reads, while red shows writes. The distribution of page accesses across an extremely large spectrum, as well as the different access patterns motivates the need for intelligent page placement when dealing with heterogeneous memories. `streamcluster` has been omitted due to space constraints.

10,630 total unique pages are accessed, with 10,628 read and 10,430 written, so nearly all data is consistently read and written back. Compare this to `fluidanimate` or `facesim` where nearly every page is written to, but 18% and 46% of these pages are not read during the execution of the respective applications. These differing access patterns not only emphasize the advantage of application/programmer guidance in identifying page use, but also the need for intelligent page placement.

Figure 3 shows the page access distribution for both reads and writes. The x-axis consists of bins of access frequency of a given page on a log scale, while the y-axis shows the number of pages in each bin. Here again we see much different access patterns of pages across benchmarks. Benchmarks such as `dedup` and `fluidanimate` have a large number of pages with very similar access counts, indicated by the sparse but tall spikes in the graphs, while `facesim` and `blackscholes` have wider distributions of page access frequencies. This is further evidence of the aid that applications can be in identifying the access patterns of pages.

Because we wanted to evaluate the entire execution of the benchmarks instead of taking a computable section of the memory traces, we were unable to use a full memory system simulator such as NVMain [23] to estimate the power and performance on different memory topologies. Using energy consumption estimates for upcoming versions of different memory technologies [24], we calculated the power consumption of the workloads using the collected memory traces. Figure 4 shows representative data from streamcluster. This ignores the effect of the CPU caches, since the same application will access the same data across all memory technologies, and instead uses the estimated 2017 joule/bit data found in Moreau for all memory accesses.

With the exception of DRAM and FeRAM, all the candidate SCM technologies have asymmetric energy consumption for reads and writes. With a goal of minimizing energy consumption, DRAM+PCM data uses simple inclusive caching to service reads through PCRAM and writes on DRAM, both allowing each technology to operate in its lowest power region

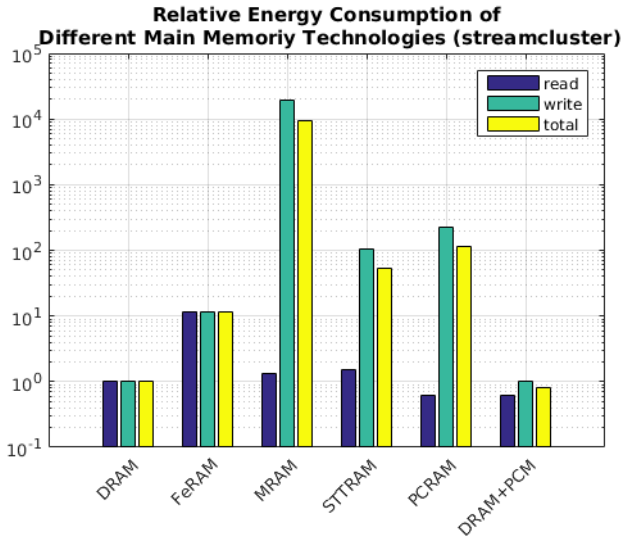

Fig. 4. This shows the energy consumption of different types of memory on the streamcluster workload relative to DRAM.

as well saving the endurance of writes to PCM.

B. Introduction of Multiple Regions on Performance

Moving to Soft2LM implementation, we will first examine the partitioning of physical memory in the kernel. In this section, we show the low cost of an active Soft2LM kernel in both in overall benchmark runtime as well as time spent executing kernel code, versus the unmodified Linux kernel v.3.14.39. The basic split-level memory management system does not migrate pages, it simply allocates pages in the lowest-latency region of memory available, and uses vmscan to handle the aging and removal of pages. Our results, Figure 5 show that there is very little performance regression by the additional codepath.

The mean real slowdown with unoptimized code was 1.5% (0.9857), with the worst regression being 8.4% in streamcluster, and the biggest improvement being 3.6% in dedup. There

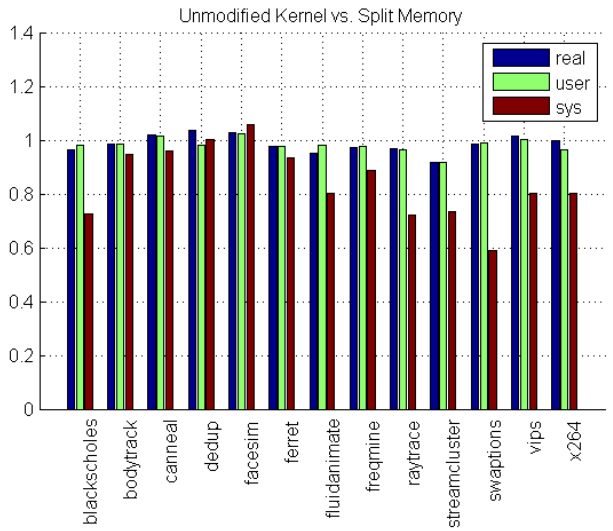

Fig. 5. Speedup of an unmodified (stock) kernel to the Soft2LM kernel. We compare the real, user, and system times (described in IV.A) to determine the cost of longer codepaths.

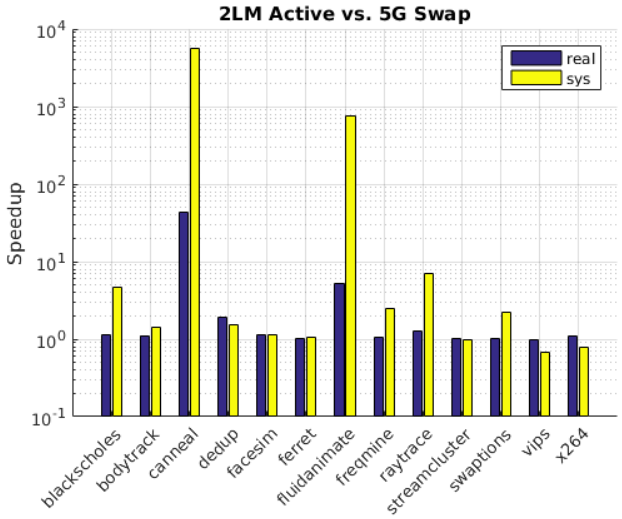

Fig. 6. Data comparing the system with two active regions of memory, one 1GB and one 5GB versus a system with 1 GB RAM and 5 GB RAMDISK backed swap.

was a greater slowdown in the kernel time, however we don't believe this is a great concern.

Given the added benefits of tiered memory (e.g. lower power consumption, persistence of data, etc.), there is only a minor regression in the performance of these memory-intensive benchmarks. Also, the amount of time spent in the kernel is very small compared the application. Averaged across all PARSEC benchmarks on a stock kernel, the CPU spends 153x more time executing the average application than in the kernel. In addition, compared to swapping as shown below, the performance regression is significantly smaller using Soft2LM. Finally, the current kernel modifications have not been optimized, and are currently very flexible for testing purposes.

C. Comparison to swapping to a RAMDISK

Instead of our method, a simpler use of SCM is to use it is a very fast block-storage device and swap to it. In this section we examine the benefits of using tiered-memory over swapping. Because using a traditional SSD/HDD-based swap device would be an unfair comparison with access time dominated by disk latency, we chose to create a swap device in RAM, thus only testing the actual overhead of the paging mechanism.

Mirroring the layout of tiered memory, we created a 5 GB swap file in DRAM. While we expected much better performance from tiered memory since the this code does not require page fault-triggered remapping, our data show that using the kernel's paging system drastically degrades performance, no matter how low the latency of the underlying device.

We ran the benchmarks on two separate swap configurations, one with 1 GB RAM and 5 GB of RAM disk-backed swap which matches the layout of the Soft2LM kernel, and one with 2 GB RAM and 4 GB RAMDISK-backed swap.

The results for a 5 GB swap device are shown as Figure 6. Note that because the speedup over the ramdisk is so large for some benchmarks, the speedup is displayed using a log y-axis. The mean and median speedup the real execution time of benchmarks using tiered-memory over a 5 GB Ramdisk are 4.7x and 1.116x respectively, while the kernel code showed a 1.53x median speedup. The performance of `canneal` was incredibly degraded when swapping, showing a 97.8% decrease in performance (43x speedup when using Soft2LM). `dedup` and `fluidanimate` both saw speedups of 1.5x and 5.2x respectively. Both of these benchmarks have rather large memory footprints as shown in Table II Both `vips` and `x264` showed a marked increase in time spent in the kernel (47% and 24% respectively), most likely due to the smaller data sets not needing to be swapped to disk, thus eliminating the cost of the swapping portion of the kernel code. It is important to note that even though there was an increase in kernel latency in these two benchmarks, the overall performance of the benchmark was not notably changed. In the remainder of benchmarks, the time spent in the kernel was significantly decreased when using two regions of memory versus swapping, with a mean speedup of 1.5x. Only `vips` had a 0.8% overall slowdown due to the higher amount of time spent in kernel code.

Because a number of these benchmarks performed so badly with a 5 GB swap device, especially `canneal` and `fluidanimate`, we reran the benchmarks with 2 GB main memory and a 4 GB swapfile to test more modest swapping. These results are shown in Figure 7. We can see that with more DRAM, improvement using the tiered-memory manager is decreased, however there is still on average a 6.3% real time speedup over swapping, with a 13% speedup in kernel code.

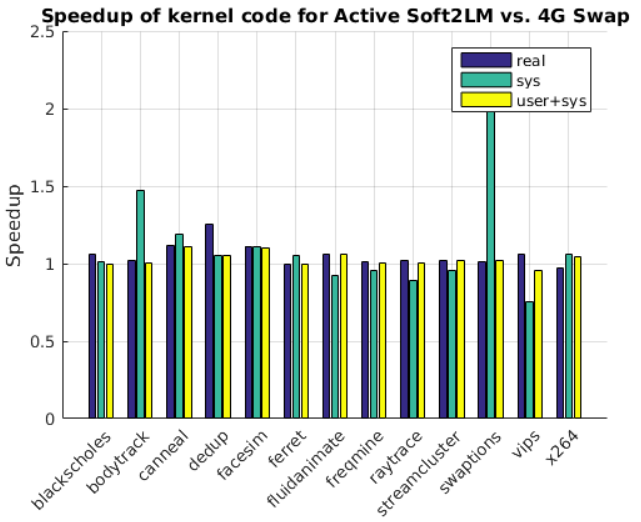

Fig. 7. Data comparing the system with two active regions of memory, one 1GB and one 5GB versus a system with 2GB RAM and 4GB RAMDISK backed swap.

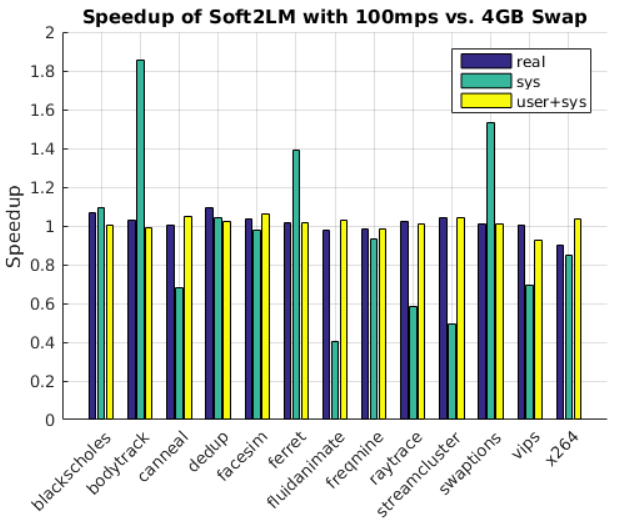

Fig. 8. Data showing the speedup when migrating 100 pages per second versus swapping.

D. Active Migration with Benchmark

While using two regions of memory shows an improvement over any amount of swapping, we also wanted to evaluate heavy migration on top of managing multiple regions of memory. Further tests were conducted while migrating 100 and 1000 pages every second, not due to memory pressure based upon a threshold, but to simply maximize the amount of migrations in order to determine the expected overhead of significant migrations. By moving pages from DRAM to SCM even when not under memory pressure, we allow new pages to be allocated in DRAM, thus implementing a basic system of page caching: new, hotter pages, are allocated first in DRAM, then as they cool are moved to SCM. The results of moving 100 pages per second are shown in Figure 8.

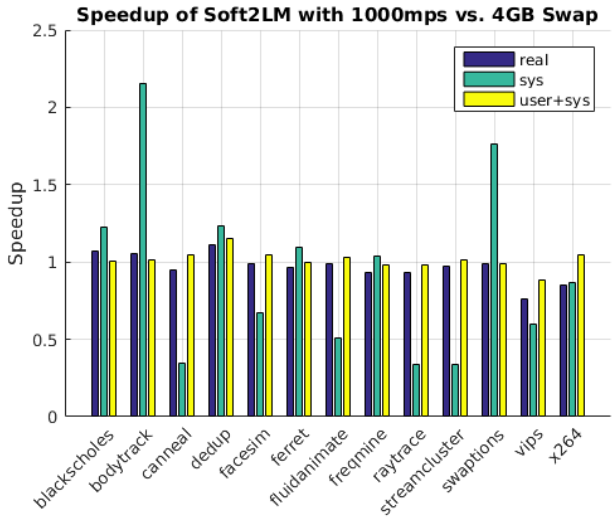

Fig. 9. Data showing the speedup when migrating 1000 pages per second versus swapping.

Migrating 100 pages per second is, given 4 KB pages, 400 KB of migrations per second. Compared to the less aggressive 4 GB swapping case, tiered migration made a slight average benchmark improvement of 1.8%, while exhibiting a 6.7% slowdown in kernel code execution. Some benchmarks such as `bodytrack`, `ferret`, and `swaptions` had significant improvements in kernel execution times.

Putting even more pressure on the migration system, we increased the number of pages to migrate per second to 1000. This very aggressive migration averaged 75,000 pages moved per benchmark run, which with 4 KB pages is 300 MB moved over the course of each benchmark. Comparing this migration number to the memory footprints of the benchmarks, it's almost certain that a large number of these applications pages were moved from under it, with almost zero effects on the runtime performance of the programs. Because we are forcing migrations without being under memory pressure, not only are the inactive lists cleared out, but the active lists are heavily scanned as well.

The data comparing 1000 page migrations per second versus a 4 GB swapfile is shown as Figure 9. These results are very similar to those in the 100 migration case (Figure 8). Even by increasing the number of migrated pages by 10x, Soft2LM still improves over this limited swapping by an average of 0.7%, though spending nearly 6.3% more time in the kernel. The performance scales well, and when in some cases moving the entire benchmark's dataset, performance is still favorably comparable to 4GB RAMDISK-based swapping, to say nothing of its massively better performance against 5 GB swapping.

V. Summary

In this paper, we have demonstrated the feasibility of a software-controlled memory controller for heterogeneous memories. We built the system on a modern version of the Linux kernel, ran it on real hardware, and tested it using a well-

respected suite of memory-intensive benchmarks. The system is robust, having been tested with heavy memory usage as well as hundreds of thousands migrations. Our results show that the tiered memory manager by itself is favorably comparable to a stock kernel, and greatly outperforms the swapping system even when using a low-latency DRAM-disk as a swapfile. Results further show that even when forcing the migration of tens of thousands of pages and hundreds of megabytes over the course of the benchmark, it still compares favorably to swapping. This system also introduces an API which allows higher-level abstractions to actively control the allocation and migration of pages, opening the door to more intelligent methods of memory management in both application and operating system.

ACKNOWLEDGMENT

This work was supported under NSF Grant CMMI 1538877.

REFERENCES

[1] R. F. Freitas, "Storage Class Memory: Technology, Systems and Applications," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 985–986. [Online]. Available: http://doi.acm.org/10.1145/1559845.1559961

[2] 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*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 24–33. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555760

[3] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: A Hybrid PRAM and DRAM Main Memory System," in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC '09. New York, NY, USA: ACM, 2009, pp. 664–469. [Online]. Available: http://doi.acm.org/10.1145/1629911.1630086

[4] O. Zilberberg, S. Weiss, and S. Toledo, "Phase-change Memory: An Architectural Perspective," *ACM Comput. Surv.*, vol. 45, no. 3, pp. 29:1–29:33, Jul. 2013. [Online]. Available: http://doi.acm.org/10.1145/2480741.2480746

[5] C. Lameter, "NUMA (Non-Uniform Memory Access): An Overview," *Queue*, vol. 11, no. 7, pp. 40:40–40:51, Jul. 2013. [Online]. Available: http://doi.acm.org/10.1145/2508834.2513149

[6] D.-J. Shin, S. K. Park, S. M. Kim, and K. H. Park, "Adaptive Page Grouping for Energy Efficiency in Hybrid PRAM-DRAM Main Memory," in *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, ser. RACS '12. New York, NY, USA: ACM, 2012, pp. 395–402. [Online]. Available: http://doi.acm.org/10.1145/2401603.2401689

[7] M. Lee, V. Gupta, and K. Schwan, "Software-controlled Transparent Management of Heterogeneous Memory Resources in Virtualized Systems," in *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, ser. MSPC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:6. [Online]. Available: http://doi.acm.org/10.1145/2492408.2492416

[8] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, "Transparent Hardware Management of Stacked DRAM As Part of Memory," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 13–24. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.56

[9] 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 *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 1–12. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.63

[10] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor, and S. W. Keckler, "Page Placement Strategies for GPUs Within Heterogeneous Memory Systems," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 607–618. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694381

[11] C. Li, Y. Yang, H. Dai, S. Yan, F. Mueller, and H. Zhou, "Understanding the tradeoffs between software-managed vs. hardware-managed caches in GPUs," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2014, pp. 231–242.

[12] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page Placement in Hybrid Memory Systems," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011, pp. 85–95. [Online]. Available: http://doi.acm.org/10.1145/1995896.1995911

[13] J. Meza, Y. Luo, S. Khan, J. Zhao, Y. Xie, and O. Mutlu, "A Case for Efficient Hardware/Software Cooperative Management of Storage and Memory," *Proceedings of the Workshop on Energy-Efficient Design (WEED)*, Jun. 2013. [Online]. Available: http://repository.cmu.edu/ece/370

[14] H. Seok, Y. Park, K.-W. Park, and K. H. Park, "Efficient Page Caching Algorithm with Prediction and Migration for a Hybrid Main Memory," *SIGAPP Appl. Comput. Rev.*, vol. 11, no. 4, pp. 38–48, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2107756.2107760

[15] H. Yoon, J. Meza, R. Ausavarungnirun, R. Harding, and O. Mutlu, "Row buffer locality aware caching policies for hybrid memories," in *2012 IEEE 30th International Conference on Computer Design (ICCD)*, Sep. 2012, pp. 337–344.

[16] M. K. Qureshi and G. H. Loh, "Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. Washington, DC, USA: IEEE Computer Society, 2012, pp. 235–246. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2012.30

[17] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management," *IEEE Comput. Archit. Lett.*, vol. 11, no. 2, pp. 61–64, Jul. 2012. [Online]. Available: http://dx.doi.org/10.1109/L-CA.2012.2

[18] L. man page, *madvise(2) Linux Programmers's Manual*, GNU, April 2014.

[19] M. R. Jantz, C. Strickland, K. Kumar, M. Dimitrov, and K. A. Doshi, "A Framework for Application Guidance in Virtual Memory Systems," in *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '13. New York, NY, USA: ACM, 2013, pp. 155–166. [Online]. Available: http://doi.acm.org/10.1145/2451512.2451543

[20] C. Cantalupo, V. Venkatesan, J. R. Hammond, and S. Hammond, "User extensible heap manager for heterogeneous memory platforms and mixed memory policies," 2015.

[21] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 72–81. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454128

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

[23] M. Poremba, T. Zhang, and Y. Xie, "Nvmain 2.0: A user-friendly memory simulator to model (non-)volatile memory systems," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 140–143, July 2015.

[24] M. Moreau, "Estimating the energy consumption of emerging random access memory technologies," Ph.D. dissertation, Institutt for elektronikk og telekommunikasjon, 2013. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:no:ntnu:diva-22566