BATMAN: Maximizing Bandwidth Utilization of Hybrid Memory Systems

Chiachen Chou, Aamer Jaleel, Moinuddin K. Qureshi

CARET Lab
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332-0250

TR-CARET-2015-01
March 9, 2015

This page is intentionally left blank.

BATMAN: Maximizing Bandwidth Utilization of Hybrid Memory Systems

Chiachen Chou* Aamer Jaleel† Moinuddin K. Qureshi

Contact: Chiachen Chou at cc.chou@ece.gatech.edu

***Abstract*—High bandwidth memory technologies such as HMC, HBM, and WideIO provide 4x-8x higher bandwidth than commodity DRAM while maintaining similar random access latency. The limited capacity of such high bandwidth memories require them to be architected in conjunction with traditional DRAM. Such a system would designate high bandwidth memory as Near Memory (NM) and commodity DRAM as Far Memory (FM), and rely on techniques that try to maximize NM hit rate. We show that the conventional approach of optimizing for NM hit rate is inefficient as it does not maximize the overall available system bandwidth. For example, when the application working set entirely fits in NM, all requests are serviced only by NM and the bandwidth of FM remains unutilized. We show that optimizing for overall system bandwidth, instead of hit rate of NM, can significantly improve system performance.**

We observe that performance can be maximized when the accesses to NM and FM are in proportion to the bandwidth of each memory. Our proposal, *Bandwidth Aware Tiered-Memory Management (BATMAN)*, can achieve this access distribution at runtime, without relying on prior knowledge of application access pattern. For systems that do not perform data migration between NM and FM, BATMAN divides the working set randomly between NM and FM to guarantee the target access distribution. For systems that perform data migration between NM and FM (such as OS-based page migration or hardware-based cache line migration), BATMAN achieves the target access distribution by monitoring the distribution of accesses at runtime and uses this information to regulate the rate of data migration. Our evaluations on a system with 4GB of NM and 32GB of FM shows that BATMAN improves performance by as much as 40% (and on average, 10%, 10%, and 22% for systems with no data migration, OS-based page migration, and hardware cache, respectively), while requiring less than 12 bytes of storage overhead.

I. Introduction

Memory technologies such as Hybrid Memory Cube (HMC), High Bandwidth Memory (HBM), and Wide I/O (WIO) [1, 2, 3] cater to growing demands for high memory bandwidth. Although these memory technologies have high memory bandwidth, they have similar memory array access latency, as shown in Figure 1. These modules are projected to have limited capacity (1GB-8GB) compared to commodity DRAM (i.e. DDR3, and DDR4 [4, 5]). Therefore, they are unable to completely replace commodity DRAM. Instead, these high bandwidth memory technologies are used in conjunction with commodity DRAM to form what is referred to as a *Hybrid-Memory System* or a *Tiered-Memory System*.

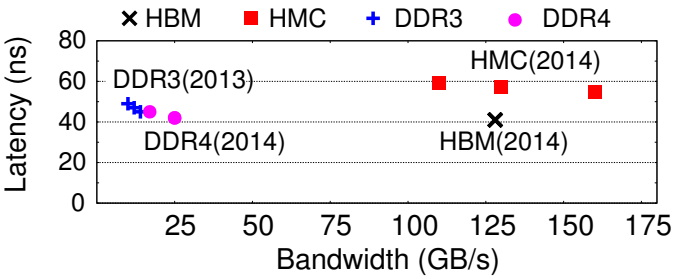

Fig. 1. Latency and bandwidth of conventional and emerging DRAM technologies (data obtained from [1, 2, 3, 4, 5])

A hybrid memory system typically consists of low-capacity, high-bandwidth memory, which we refer to as *Near Memory (NM)*, and high-capacity low-bandwidth commodity DRAM memory, which we refer to as *Far Memory (FM)*. The NM in the hybrid memory systems can be configured either as a hardware-managed DRAM cache [6, 7, 8, 9, 10, 11] or part of the OS-visible main memory [12, 13, 14, 15, 16]. In either case, the system tries to maximize the fraction of total memory requests that get satisfied by the NM. For example, if NM is used as hardware-managed cache, an access to the line in FM will install the requested cache line in NM, with the objective that subsequent references to this line will get satisfied by NM (improving cache hit rate). Similarly, if the NM is architected as OS-visible main memory, then the OS will try to maximize the number of NM pages allocated to the given application. The underlying assumption in both cases is that servicing the working set from high bandwidth NM (and not low bandwidth FM) will provide the highest system performance. We show that these conventional approaches to maximize the hit rate of the NM are inefficient, because they may not fully utilize the entire system bandwidth. For example, when the working set of the application fits in the NM, the bandwidth of the FM remains unutilized. We show that optimizing for overall system bandwidth, instead of the hit rate of NM, can result in much higher system performance.

*†Part of this work was done when Chiachen Chou was a summer intern at Intel. Also, Aamer Jaleel was with Intel during this work.

‡This technical report is made public available on March 9, 2015. Please contact Chiachen Chou at cc.chou@ece.gatech.edu for any questions.

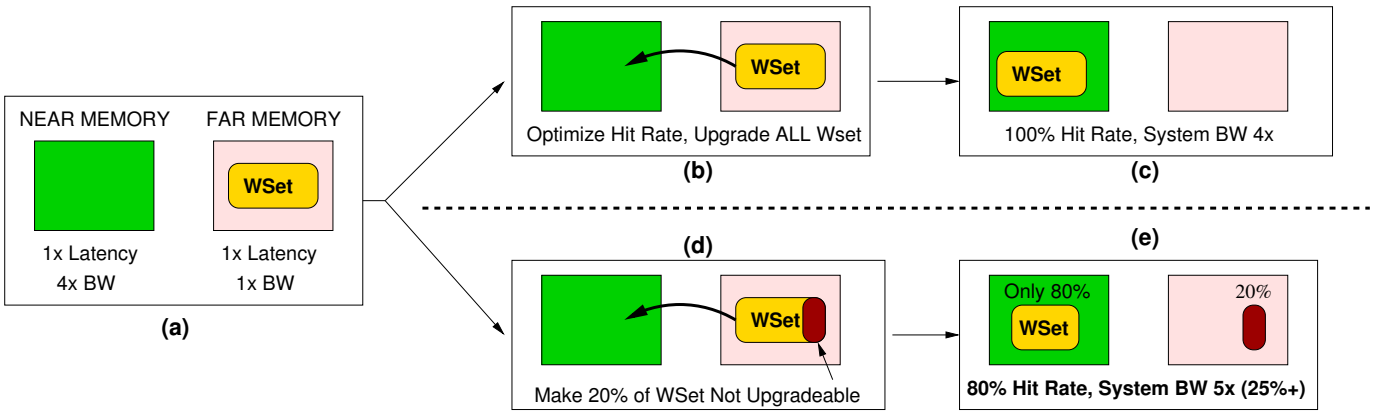

Fig. 2. Optimizing for Hit Rate versus for System Bandwidth. (a) a system with near memory and far memory, both having the same latency but near memory has 4x the bandwidth, (b) and (c) traditional systems that try to optimize for hit rate gives up to 4x system bandwidth, (d) and (e) explicitly controlling some part of the working set to remain in far memory results in up to 5x system bandwidth.

We explain the effectiveness of optimizing for system bandwidth instead of the hit rate in NM with an example. Figure 2 (a) shows a hybrid memory system, where the NM and FM have the same latency, but the NM has 4X the bandwidth of FM. Without loss of generality, we consider an application whose working set is small enough to fit in both NM and FM. Figure 2(b) and (c) shows the traditional approach which tries to maximize the hit rate of NM. On an access to the data in FM it would move that data element to NM, as shown in Figure 2(b). Over time, the entire working set of the application would move to the NM, and then all of the accesses will serviced only by the NM. The FM would no longer receive any accesses, and the overall system bandwidth is determined by NM, becoming at most 4x, as shown in Figure 2(c).

Now, consider an alternative approach that tries to maximize the overall system bandwidth. For example, we could identify the part of the working set that is responsible for 20% of memory accesses for the given application and mark it ineligible for upgrade from FM to NM. An access to such a data item in FM would not move the data from FM to NM. Therefore, in steady state, NM would be servicing, on average, 80% of the the accesses and FM would be servicing 20% of the accesses. Thus, the overall system bandwidth would now be 5x, with the memory traffic balanced between NM and FM. Thus, explicitly controlling the fraction of memory accesses that get serviced by NM and FM can lead to higher aggregate system bandwidth, and thus system performance, even if it means a lower hit-rate in the NM. We propose *Bandwidth Aware Tiered-Memory Management (BATMAN)*, which aims at maximizing the overall memory system bandwidth.

To obtain the highest performance BATMAN must split the accesses between NM and FM in proportion to the bandwidth of each memory. For a system that does not perform data migration between NM and FM, the target access division between NM and FM can be achieved by controlling the number of pages allocated to the NM, such that they account for only a given percentage of total memory accesses (say 80%). Unfortunately, page access count is typically not available, and generating access counts per page at runtime tends to be quite complex [16]. Ideally, we want to achieve the target split without relying on the page access counts. BATMAN leverages a key insight that if the application's working set is *randomly* partitioned in proportion to the memory bandwidth, the accesses to NM and to FM will also get split proportionally. In other words, if NM contributes 80% of the total bandwidth in the memory system, then retaining 80% of the *randomly* selected pages from the working set in NM should provide close to 80% of the accesses (regardless of the variation in page access pattern, as long as such variation is not pathologically high). We demonstrate that the proposed randomized partitioning of the working set with BATMAN is highly effective at obtaining the target access rate using both analytical model and extensive experimental evaluation.

We also analyze BATMAN for systems that configure NM as an OS-visible main memory, where OS can migrate pages between NM and FM [17, 16]. For such systems, BATMAN achieves the target access rate by explicitly monitoring the number of accesses serviced by NM at runtime, and uses this information to dynamically regulate the rate and the direction of migration between NM and FM. We show that our proposed design, which incurs an overhead of only 12 bytes, is highly effective at achieving the target access rate in NM.

Finally, we also study BATMAN for an OS-transparent system that uses NM purely as a hardware-managed cache. In this scenario, BATMAN monitors the access rate (including reads and writes). If the access rate exceeds a predetermined threshold, BATMAN dynamically disables portion of the cache to reduce the rate to the desired level. Doing so ensures that a given fraction of the working set always gets serviced by the FM, maximizing overall system bandwidth and performance.

This paper makes the following contributions:

- We identify that optimizing for overall system bandwidth, instead of hit rate of NM, can give higher performance and propose BATMAN to distribute accesses between NM and FM in proportion to each memory's bandwidth.

- We demonstrate BATMAN for a system that statically maps pages between NM and FM. We show, both analytically and experimentally, that our randomized partitioning of the working set is quite effective in obtaining the target access rate for applications that would fit in NM.
- We design BATMAN for systems that does data migration between NM and FM (either OS-based page transfers or hardware-based cache line transfers) and show that controlling the data migration rate based on runtime activity is effective at obtaining the target access rate.

We evaluate BATMAN on a 16-core system with 4GB of NM (at 100 GBps) and 32GB of FM (at 25 GBps). The average speedup of BATMAN is 10% for a system without data migration, 10% for a system that does OS-based page migration, and 22% for a system that uses NM as a cache.

II. Background and Motivation

Emerging high bandwidth memory technologies such as HBM, HMC, and WideIO [1, 2, 3, 4, 5] can help high-performance systems overcome the memory bandwidth wall. These technologies provide a 4x-8x bandwidth improvement over commodity DDR memories while maintaining a similar random access latency. Unfortunately, manufacturing and cost constraints limit these high bandwidth memory technologies to small storage capacity (few GB), therefore, high bandwidth memory technologies are commonly used in conjunction with commodity DRAM. In such a system, the low-capacity high-bandwidth memory can be deemed as the *Near Memory (NM)* and the commodity memory can be deemed as the *Far Memory (FM)*. The NM can be configured either as a OS-managed main memory or as a hardware-managed DRAM cache. In either case, the critical factor that determines the effectiveness of incorporating the NM in the system is the percentage of accesses that get serviced by the Near Memory.

A. Conventional Wisdom: Optimize for Hit Rate

When the application working set is larger than the available NM capacity, both the NM and FM data buses provide data to the processor. However, when the application working set is smaller than NM capacity, conventional approach try to retain almost all of the data in NM, as memory request serviced from NM are serviced at higher bandwidth. Thus, conventional approaches try to maximize the hit rate of the NM. When the application working set fits into the available NM capacity, only the NM data bus is utilized while the the FM data bus remains idle. So, conventional designs that try to optimize for hit rate come at the expense of leaving system bandwidth under utilized. Ideally, for maximizing the overall system bandwidth and performance, we would like to make use of the bandwidth available from both the NM and FM.

B. Our Insight: Optimize for System Bandwidth

We can ensure that the bandwidth of both NM and FM gets used by regulating the percentage of accesses that get serviced by the NM. We corroborate this hypothesis experimentally. We study a set of memory intensive STREAM [18] benchmarks with a memory system configuration similar to that of Figure 2 (experimental methodology is in Section III). As the working set of the streaming benchmarks is less than the capacity of the NM, the baseline configuration services all the accesses of the STREAM benchmarks entirely from NM. We conduct a sensitivity study where we control the percentage of accesses that are serviced by the NM by explicitly allocating part of the working set in FM. Figure 3 shows the speedup compared to the baseline system as the the fraction of working-set serviced from NM is increased form 10% to 100%. We study bandwidth ratio of 2X, 4X, and 8X between NM and FM. For all systems, the peak performance occurs much earlier than 100%, validating our hypothesis.

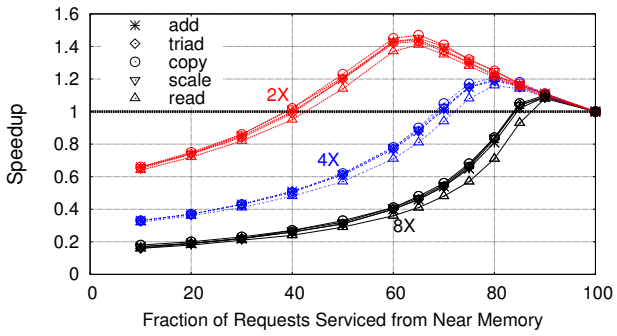

Fig. 3. Speedup versus Service Rate of Near Memory. Note the peak performance occurs much earlier than 100% (baseline system).

When the ratio of bandwidth of NM to FM is 4x, we get a peak performance of 1.2x, which is close to the ideal speedup of 1.25x that we expect by improving the system bandwidth by 25% (system bandwidth increases form 4x to 5x). A similar observation applies to the systems with 2x and 8x bandwidth ratios. Thus, there is greater potential for improvement by optimizing for system bandwidth when FM accounts for a significant fraction of the overall system bandwidth.

C. Determining the Best Distribution of Accesses

When the NM has 2x the bandwidth of FM, we observe that the peak performance occurs when the hit rate of NM is approximately 66%, meaning NM services $\frac{2}{3}$ of the accesses and FM services $\frac{1}{3}$ of the accesses. Similarly, when the NM has 4x the bandwidth, the peak performance occurs when the hit rate of the NM is approximately 80%: the NM services $\frac{4}{5}$ of the accesses and FM services the remaining $\frac{1}{5}$ of the accesses. For 8X NM bandwidth, the peak performance occurs when $\frac{8}{9}$ of accesses are serviced by NM and $\frac{1}{9}$ by FM. These results indicate that for the highest overall system performance, the accesses must be distributed between Near Memory and Far Memory in proportion to the relative bandwidth offered by the respective memory subsystems.

D. Goal: Simple and Effective Access Distribution

We seek a mechanism that can distribute the memory accesses between NM and FM, in proportion to the bandwidth

ratio of each memory. If we had page access counts for all pages in a given working set, we could decide which page goes in NM to obtain the desired access rate in NM. Unfortunately, page access counts either require profile information or a runtime mechanism that can be complex. Ideally, we want our mechanism to be effective without any prior knowledge of the application access pattern. To that end, we propose *Bandwidth Aware Tiered-Memory Management (BATMAN)*. We develop BATMAN for systems that do not perform data migration between NM and FM (Section IV), for systems that perform OS-based page migration between NM and FM (Section V), and for systems that use NM as a hardware-managed cache (Section VI). We first describe our experimental methodology before describing our solutions.

III. Experimental Methodology

A. System Configuration

We model a 16-core system using a detailed event-driven x86 simulator. The core parameters (Table I), cache hierarchy organization and latencies are loosely based on the Intel Core i7 processor [19]. Each core is a 4-wide issue out-of-order processor with a 128-entry ROB that is running at 3.2GHz. The memory system contains a three-level cache hierarchy (L1 and L2 are private, and L3 is shared between all the cores). All levels in the cache hierarchy use 64B line size.

TABLE I
Baseline System Configuration

Processors	
Number of Cores	16
Frequency	3.2GHz
Core Width	4 wide out-of-order
Prefetcher	Stream Prefetcher
Last Level Cache	
Shared L3 Cache	16MB, 16-way, 27 cycles
Near Memory (Stacked DRAM)	
Capacity	4GB
Bus Frequency	800MHz (DDR 1.6GHz)
Channels	8
Banks	16 Banks per rank
Bus Width	64 bits per channel
tCAS-tRCD-tRP-tRAS	36-36-36-144 CPU cycles
Far Memory (Commodity DRAM)	
Capacity	32GB
Bus Frequency	800MHz (DDR 1.6GHz)
Channels	2
Banks	16 Banks per rank
Bus Width	64 bits per channel
tCAS-tRCD-tRP-tRAS	36-36-36-144 CPU cycles

Our heterogeneous memory subsystem consists of 4GB of Near Memory (NM) using HBM technology [2] and 32GB of Far Memory (FM) using conventional DDR technology [4]. Our DRAM simulator is similar to USIMM [20], which has read and write queues for each memory channel. The DRAM controller prioritizes reads over writes, and writes are issued in batches. We use Minimalist-Open page [21] for the DRAM address mapping, and open-page policy as default. We assume the same random access latency in both DRAM technologies. However, the bandwidth of NM is modeled to be higher than FM. In our baseline system, NM has 4x bandwidth of FM (4X channel). A sensitivity study to bandwidth ratio is presented in Section VII.

We model a virtual memory system to perform virtual to physical address translations. We assume 4KB page sizes. We study three system configuration: First, an OS-visible NM system that does not support page migration between the NM and FM (Section IV); Second, OS-visible NM system that supports page migration between NM and FM (Section V); Third, a system where NM is configured as a hardware-managed L4 cache (Section VI). For the first system, the default page mapping policy is *NM-First*, which allocates the pages first in NM and only when it runs out of NM capacity, it allocates the pages in FM. For the second system, we start with random page mapping; however, on access to a page in FM, the system performs page migration [17, 16] from FM to NM, thus optimizing for data locality. For the third system, we configure the NM as a direct-mapped *Alloy Cache* [9] with cache hit-miss predictors. To ensure high cache performance, the virtual memory system uses page coloring to reduce conflict misses in the direct-mapped cache.

B. Workloads

We use Pin and SimPoints [22, 23] to execute one billion instructions each from various benchmarks suites, including SPEC CPU2006, Stream [18], and High Performance Computing (HPC) workloads. Table II shows the characteristics of the 20 workloads used in our study. We evaluate our study by executing benchmarks in rate mode, where all sixteen cores execute the same benchmark. As the effectiveness of our scheme depends on the relative size of the application working set with respect to the capacity of NM, we divide the SPEC applications into two categories: Applications whose working set is smaller than 256MB are categorized as SPEC_S, and SPEC_L otherwise.

C. Figure of Merit

1) Speedup:: We measure the total execution time as the figure of merit. As we run the workloads in rate mode, the difference in execution time of individual benchmark within the workload is negligibly small. Speedup is reported as the ratio of execution time of the baseline to the given scheme.

2) NM Access-Rate:: We also report the access rate of NM, defined as the number of accesses to NM divided by the total number of memory access to both NM and FM. Each number includes reads and writes.

IV. BATMAN for Systems with No Migration

In this section, we consider a system that consists of NM and FM, and there is no data migration between NM and FM (systems that support migration are analyzed in subsequent sections). For such a system, the decision that determines the bandwidth utilization of the NM is the page mapping policy. For example, a *NM-Agnostic* policy may allocate pages

TABLE II
WORKLOAD CHARACTERISTICS (FOR SINGLE CORE).

Category	Name	L3 MPKI	Footprint(MB)
HPC	hpc1	82	134
	hpc 2	48	81
	hpc 3	46	46
	hpc 4	43	132
	hpc 5	30	120
SPEC_L	milc	65	428
	lbm	64	399
	Gems	53	727
	mcf	43	1186
	bwaves	39	423
SPEC_S	soplex	64	54
	omnetpp	50	140
	libq	48	32
	leslie	43	78
	astar	25	37
STREAM	add	83	229
	triad	73	229
	copy	71	152
	scalar	64	152
	read	43	152

randomly to any location in memory. Unfortunately, given that our NM is 8x smaller than FM (4GB vs. 32GB), this policy will use NM for only 11% (one-ninth) of the pages, even if the working set of the application is small enough to fit in NM. An alternative policy, *NM-First* can optimize for NM bandwidth by first allocating the pages in NM, and only when the capacity of NM gets exhausted then allocating the pages in FM. NM-First tries to maximize the hit rate of NM by ensuring that the application receives as many pages from NM as possible. When the application working set fits in the NM, this policy will ensure that all of the accesses get serviced by NM, however the bandwidth of FM will remain unutilized.

To maximize for system bandwidth we can regulate the number of pages that get allocated to the NM to ensure that only a target fraction of accesses go to NM. In our baseline, NM has 4x the bandwith of FM, so our *Target Access Rate (TAR)* for NM is 80%. Ideally we want the NM to service only 80% of the accesses and the FM to service 20% of the accesses. We show how our proposed implementation of BATMAN can effectively achieve the TAR of 80% in NM.

A. BATMAN: Randomized Partitioning of Working Set

Memory accesses can exhibit significant non-uniformity across pages. So, a given fraction of total pages does not always result in the same fraction of all memory request getting serviced from those pages. For example, if 90% of the accesses came from only 10% of the pages, placing the 90% cold pages in NM would give a hit rate of only 10% for NM. If we had page access counts available, we could allocate an exact number of pages in NM such that the total accesses from those set of pages equals TAR. Unfortunately, the access counts per page are not typically available and obtaining them at runtime tends to be complex.

Ideally, we want BATMAN to be effective without requiring any information on the exact access pattern of a workload. BATMAN achieves this by leveraging the key insight that if we randomly partition the working set between NM and FM, in proportion to the bandwidth of each memory, the access rate to each such partition can also be expected to be in proportion to the number of allocated pages. This works well when the working set of the application can fit in NM, as we have leeway in deciding whether the page goes in NM or FM. We demonstrate the effectiveness of BATMAN at achieving the TAR with an analytical model and extensive experimental evaluation, even if BATMAN does not have any explicit information about the access pattern.

B. Analytical Model for Randomized Partitioning

Let the application consist of N pages, where the average number of accesses per page is denoted by μ and the standard deviation in the number of accesses per page is denoted by σ. Let CoV denote the coefficient of variation in accesses per page, defined as the ratio of σ to μ, and serves as an indicator in variability in accesses across different pages. The total number of memory accesses in the application, which we denote as $Access_{Tot}$, is given by Equation 1.

$$Access_{Tot} = N\mu \tag{1}$$

Let the target access rate in the NM be α ($\alpha = 0.8$ in our case). Let each page be randomly allocated to NM with probability α and to FM with probability $(1 - \alpha)$. Let the number of accesses serviced by NM be $Access_{NM}$. Given that number of randomly allocated pages to NM is large ($\alpha \cdot N$), we can use the *Central Limit Theorem* [24] to approximate $Access_{NM}$ as a Gaussian distribution, regardless of the exact distribution of the access pattern of the original workload. The expected value of $Access_{NM}$, which we denote by μ_{SumNM}, is given by Equation 2. The standard deviation of $Access_{NM}$, which we denote by σ_{SumNM}, is given by Equation 3.

$$\mu_{SumNM} = \alpha N\mu = \alpha \cdot Access_{Tot} \tag{2}$$

$$\sigma_{SumNM} = (\sqrt{\alpha N}) \cdot \sigma \tag{3}$$

The Coefficient of Variation of total number of accesses to NM, CoV_{SumNM}, can then be determined by Equation 4:

$$CoV_{SumNM} = \frac{1}{\sqrt{\alpha N}} \cdot \frac{\sigma}{\mu} = \frac{1}{\sqrt{\alpha N}} \cdot CoV \tag{4}$$

From Equation 2, we can expect the hit rate of NM to be close to α (assuming the NM is large enough to hold all the pages). From Equation 4, the variation in the total accesses goes down as the square root of the number of all the pages, which means randomized allocation can be expected to have negligibly small variation in the access rate to NM. For example, consider a hypothetical workload for which all the accesses come from only 1% of the pages. For this workload σ will be approximately 10 times the μ (CoV of 10). If we sample 1 million pages from this application, the sum will increase by a million times the average, however, the standard

deviation of the sum will increase by only a thousand times the original standard deviation. So, the CoV of the sum will reduce from 10 to 0.01. From the theory on confidence intervals, we know that the Gaussian random variables can be expected to be within 2 standard deviation of the mean with 95% confidence. So, given that CoV of the sum is only 1%, we can expect (with 95% confidence) that the access rate to NM to remain within 2% of the TAR, even though the original workload had fairly high variation in accesses per page.

Table III shows the different characteristics of the workloads we study, including N (the number of pages), μ (mean number of accesses per page), σ (standard deviation of access per page), and the 95% confidence range of access rate in NM as derived from Equation 4. For this analysis, we assume that NM is large enough to accomodate all the pages of the application (as randomize partitioning by itself cannot guarantee 80% hit rate for NM, if the NM does not have enough capacity to provide a 80% hit rate to begin with). Across all workloads, the maximum range of accesses in near memory is very close to our goal of 80%. This indicates randomized partitioning of the application working set is highly effective at obtaining the target access rate in NM.

TABLE III
BOUNDS ON NM ACCESS RATE USING ANALYTICAL MODEL.

Name	N (# Pages)	μ (Mean)	σ (Stdv.)	95% Conf. Range (%)
hpc1	34475	2,949	3,246	78.9 -81.1
hpc2	20859	2,503	3,292	78.4 -81.6
hpc3	11812	4,500	7,200	77.4 -82.6
hpc4	33849	1,380	3,376	77.6 -82.4
hpc5	30899	1,161	1,579	78.6 -81.4
milc	109597	705	180	79.9 -80.1
lbm	102391	590	362	79.7 -80.3
Gems	186166	324	154	79.8 -80.2
mcf	303798	168	1,202	77.7 -82.3
bwaves	108314	484	313	79.6 -80.4
soplex	13855	5,375	9,030	77.4 -82.6
omnetpp	35954	1,689	1,698	79.1 -80.9
libq	8200	7,149	866	79.8 -80.2
leslie	19969	2,409	2,170	78.9 -81.1
astar	9625	3,459	3,453	78.2 -81.8
add	58634	1,606	516	79.8 -80.2
triad	758650	1,404	456	79.8 -80.2
copy	39105	1,990	588	79.7 -80.3
scale	39120	1,751	579	79.7 -80.3
read	39105	1,563	79	80.0 -80.0

C. Experimental Validation of Randomized Partitioning

We also demonstrate the effectiveness of randomized partitioning at achieving the TAR using experimental evaluation. We conducted the following experiments: On a page miss, we allocate the page in NM with a probability of α (α equal to 80% in our study); otherwise, the page is allocated in FM. We set the FM such that it can hold all pages allocated to it, and measured the percentages of accesses serviced by the FM. We ran this experiments 1000 times (with a different random seed) and measure the hit rate of NM for each experiment. We obtain the average, standard deviation, maximum and minmum values from these 1000 data points. From the average and standard deviation, we computed the 95% confidence interval based on the experimental data. Figure 4 shows the 95% confidence interval range, maximum and minimum from our 1000 experiments, and the average from our experiments for each of the workloads. For all workloads, the 95% confidence interval, the maximum, and the minimum are all within 2% of the target access rate of 80%, thus providing strong evidence that randomized partitioning can help BATMAN regulate the access rate to NM.

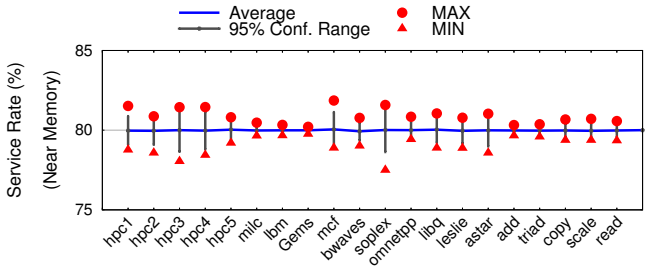

Fig. 4. Experimental validation of Randomized Partition. The solid line is the mean; the error-bar indicates the 95% confidence range based on experimental data; the dot and the triangle are the respective maximum value and the minimum value observed in 1000 trials.

D. Performance Improvement with BATMAN

Figure 5 shows the speedup of BATMAN and NM-Agnostic, both normalized to the baseline system that performs *NM-First* mapping. The bar labeled *AVG* represents the geometric mean speedup measured over all the 20 workloads. As NM-Agnostic can arbitrarily allocate pages from any location in memory, it only allocates 11% of the application pages in NM (given NM is 4GB and FM is 32GB) even if the workload could fully fit in NM. So, it significantly under utilizes the bandwidth of NM, and hence suffers a performance degradation of almost 60% compared to NM-First.

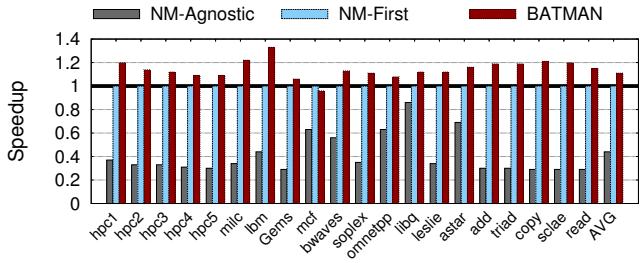

Fig. 5. Speedup of BATMAN and NM-Agnostic normalized to the baseline system which uses NM-First page allocation.

NM-First, on the other hand, tries to maximize NM bandwidth, but misses out on the FM bandwidth when the working set fits into NM. BATMAN uses randomized partitioning to explicitly control the access rate to NM to the TAR value of 80%, hence it uses the bandwidth of both NM and FM effectively, which improves the average performance by 10%.

E. Effectiveness of BATMAN at Reaching TAR in NM

BATMAN is designed to lower the Hit Rate of NM, if it exceeds the TAR value. So, it can be expected to successfully achieve TAR in NM, if the hit rate of the NM for the baseline system exceeds the TAR value, as would be the case when the working set fits in the NM. If the NM is not large enough to hold a significant fraction of working set to begin with and achieves a hit rate of less than 80% in NM even for the baseline system, BATMAN will be unable to reach the TAR, as BATMAN can only lower or retain the hit rate of NM, and is not designed to increase the hit rate of NM.

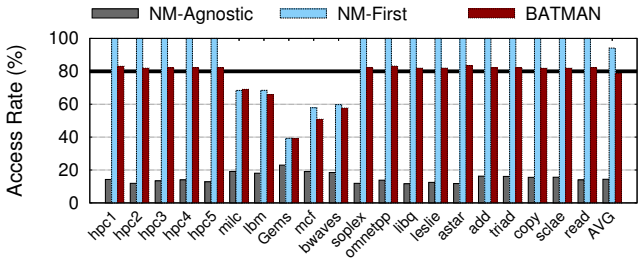

Fig. 6. Access Rate of Near Memory for NM-First (Baseline system), NM-Agnostic, and BATMAN.

Figure 6 shows the hit rate in NM for the baseline system (NM-First), NM-Agnostic and BATMAN. NM-Agnostic has as average hit rate of less than 15% for NM, as it allocates only a small fraction (11%) of all pages to the NM. NM-First has almost 100% hit rate for NM for all workloads, except the ones from the SPEC_L category (milc-bwaves). For these applications, the working set is much larger than the capacity of NM, and hence the hit rate is much less than 70%.

BATMAN achieves the TAR of 80% effectively for all the workloads, for which the hit rate for the baseline exceeds 80%. For the remaining workloads, BATMAN retains a hit rate similar to the baseline system. Overall, we conclude that BATMAN achieves the TAR, if the baseline hit rate of NM exceeds the TAR value, which is an indicator that the system is under utilizing the FM, and is the key inefficiency we are trying to address with BATMAN.

V. BATMAN for Systems with Page Migration

For systems that do not support page migration, once a page gets mapped to a particular type of memory region the mapping continues until the page gets evicted from memory. Therefore, the performance of such systems tend to be quite sensitive to the initial mapping of the pages, such as NM-First or NM-Agnostic. Allowing the pages to dynamically migrate between NM and FM [17, 16] can relieve the system from sensitivity to initial placement of page. For example, an access to FM can initiate a transfer of the accessed page to NM, which can allow subsequent references to the page be serviced by the NM. Such systems exploit the temporal and spatial locality in reference streams to maximize the hit rate in the NM. When the application working set is smaller than the capacity of NM, such migrations will eventually move the entire working set to NM and thus result in almost all of the accesses to be serviced by NM. Unfortunately, doing so would leave the bandwidth of the FM unutilized. Ideally, we want to do page migration and yet ensure that the bandwidth of both NM and FM get utilized, by ensuring that no more than a target access rate (TAR) fraction of all accesses be serviced by the NM. We show how BATMAN can be designed for a system that supports page migration in order to meet the TAR.

A. Idea: Regulate Direction of Migration to Meet TAR

We observe that the hit rate of NM depends on the ability to do page migration. Migrating pages from FM to NM increases the hit rate of NM. Similarly, downgrading recently accessed pages from NM to FM reduces the hit rate of NM. So, BATMAN can regulate the hit rate simply by monitoring the number of accesses serviced by NM, and using that information to decide the direction of page migration. If the measured access rate in NM is lower than TAR, BATMAN should try to upgrade the accessed pages from FM to NM. If the measured access rate in NM is higher than TAR, BATMAN can downgrade some of the recently accessed pages from NM to FM. This can allow BATMAN to ensure that the access rate of NM remains close to the TAR.

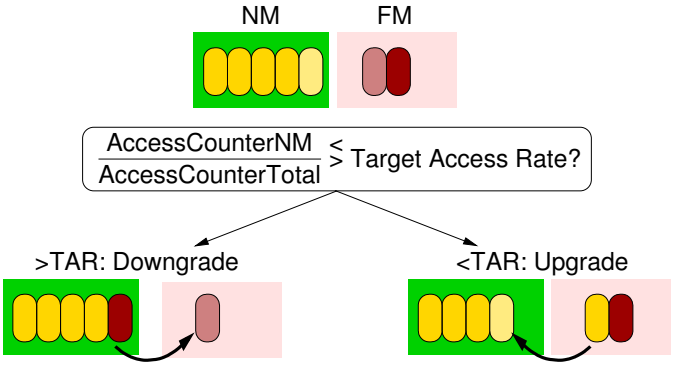

Fig. 7. Overview of BATMAN: monitoring the access rate of NM for determining the direction of migration to meet TAR in NM.

B. BATMAN: Design and Implementation

Figure 7 shows an overview of how BATMAN dynamically controls the direction of page migration to meet the TAR. Our system has NM with 4x the bandwidth, so the TAR is 80%. We provision the system to monitor the hit rate of NM and to make upgrade and downgrade decisions, as described below:

1) Structures:: BATMAN requires dynamic monitoring of the fraction of total memory accesses that get serviced by NM. BATMAN accomplishes this by using two counters: *AccesssCounterNM* and *AccessCounterTotal*, which counts the number of accesses to NM and to the system (both NM and FM), respectively. The ratio of AccessCounterNM to AccessCounterTotal is an indicator of the of the fraction of total requests serviced by NM. Both these counters are incremented on demand, prefetch, or writeback requests. We use a 16-bit

register for both counters, and when the AccessCounterTotal overflows, we halve (right shift by one) both counters.

2) Operation:: On each access, we compute the ratio of AccesCounterNM to AccessCounterTotal. If this ratio is less than TAR, we want to increase the hit rate of NM, so if the request is to FM, we upgrade the requested page from FM to NM. Similarly, if the ratio is greater than TAR, then we want to reduce the hit rate of FM. So, if the request was to a page in NM, we would downgrade the requested page from NM to FM. Regulating such downgrade and upgrade based on runtime information ensures that the access rate from FM will become close to TAR.

3) Hysteresis on Threshold:: Once the access rate of NM is close to TAR, the system can continuously switch between upgrade and downgrade. To avoid this oscillatory behavior, we provision a guard band of 2% in either direction in the decision of upgrade and downgrade. So, page upgrades happen only when the measured access rate of NM is less than (TAR-2%) and downgrades happen only when the measured access rate of NM exceeds (TAR+2%). In the intermediate zone, neither upgrades nor downgrades get performed, as the access rate to NM is already quite close to TAR.

4) Storage Overheads:: The proposed implementation of BATMAN requires a storage overhead of only four bytes (two 16-bit counters) to track the accesses to NM and the system. BATMAN leverages the existing support for page migration between NM and FM, and requires negligible modifications to the data migration logic.

C. Performance Improvement from BATMAN

Figure 8 shows the speedup from BATMAN compared to the baseline system that performs page migration between NM and FM. BATMAN improves performance of all workloads, and on average by 10%. The performance improvement comes from two factors. First, for workloads that have working set fit in the Near Memory, BATMAN effectively uses both NM and FM bandwidth, and hence has higher throughput for those workloads. For example, all the workloads in HPC, and STREAM, plus some workloads from SPEC_S (*soplex-astar*) suites belong to this group, and have higher performance. Second, for workloads form the SPEC_L category, whose working set is larger than the Near Memory, for example, *bwaves, mcf, Gems, and milc*, BATMAN reduces the number of page migration, as BATMAN only allows certain amount of pages to be upgraded to Near Memory. Since page migration consumes a lot of memory bandwidth, BATMAN helps reduce the bandwidth usage for this group of workloads, and helps obtain higher performance.

D. Effectiveness of BATMAN at Reaching TAR in NM

Figure 9 shows the hit rate of NM for the baseline system with page migration, and BATMAN. In the baseline, all workloads have close to 100% hit rate, even for the SPEC_L benchmarks suites whose working set is larger than the NM capacity. This happens because these workloads have high spatial locality within a page. For example, an accesses to FM would upgrade a page to NM, and then 15 more references could go to different lines in the page, resulting in the hit rate for NM of 15/16, or close to 94%.[1] For other applications, as their working set fits in NM, all the pages are transferred to the NM and in the steady state obtain a NM hit rate of 100%.

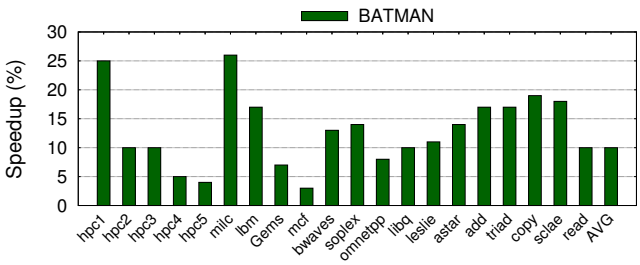

Fig. 8. Speedup for BATMAN for a system with page migration.

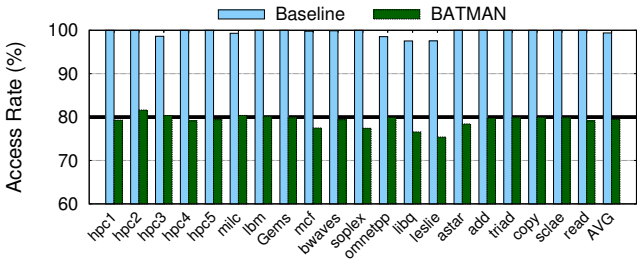

Fig. 9. Access Rate of Near Memory. BATMAN obtains a access rate close to the target access rate of NM (80%) for all workloads.

BATMAN redistributes some of the memory traffic to FM, and balance the system based on the bandwidth ratio of NM and FM. For all workloads, BATMAN effectively obtains a NM hit rate close to the target values of 80%, which helps BATMAN to balance the memory traffic, with 80% getting serviced from NM and 20% from FM. Thus, our proposed implementation of BATMAN is effective at obtaining the TAR in NM, by using a simple monitoring circuit at runtime.

VI. BATMAN for Systems with DRAM Cache

NM can also be architected as a hardware managed (L4) cache, which can serve an intermediate cache between the Last Level Cache (LLC) of the processor and the FM. Using NM as a hardware-managed cache allows ease of deployment in a software-transparent fashion. Like OS-visible near memory, DRAM caches also inefficiently utilize total effective memory system bandwidth when the majority of accesses are serviced by the DRAM cache. For example, when the application working set fits in the DRAM cache, all FM bandwidth remains unutilized as all the accesses get serviced only by the DRAM cache. Even for such a system, there is an opportunity to maximize the total available system bandwidth by dynamically regulating the fraction of working set that gets serviced by the NM-Cache.

[1]We verified that SPEC_L workloads have high spatial locality by counting the accesses before the page gets evicted from NM.

A. Idea: Regulate Hit-Rate via Partial Cache Disabling

The core idea of BATMAN for DRAM caches is to regulate the fraction of working set serviced from the DRAM cache by controlling the DRAM cache hit rate. For example, a 100% DRAM cache hit rate make FM bandwidth unusable. Instead, a 80% DRAM cache hit rate in our baseline system enables efficient utilization of total system bandwidth by servicing 20% of the requests from FM. Thus, BATMAN for DRAM caches must regulate the DRAM cache access rate to improve overall system performance. Regulating DRAM cache access rate can be accomplished by dynamically disabling some fraction of the DRAM cache until the DRAM cache access rate matches the desired target access rate. BATMAN supports disabling of DRAM cache sets by pre-selecting a subset of the DRAM cache sets to only service data from FM. Disabling the sets mitigates the overhead of doing the tag lookup to determine if the line is present in the cache.

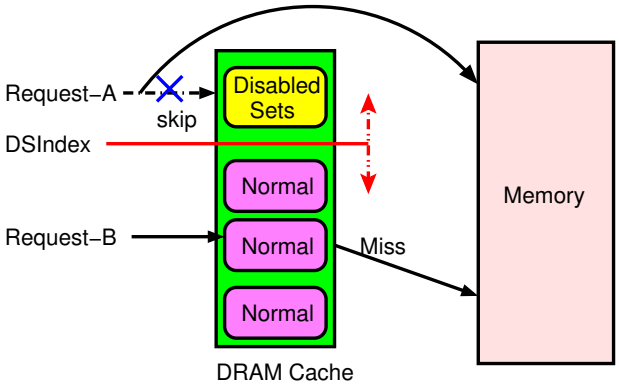

Fig. 10. BATMAN for DRAM caches. All sets at index lower than DSIndex are deemed "Disabled Sets" and do not service any cache request. DSIndex moves to modulate the hit-rate closer to the target.

B. Design of BATMAN for DRAM Cache

Figure 10 shows the overview of BATMAN with hardware cache. The key attribute of this design is the *Disabled Sets Index (DSIndex)*, which controls the fraction of cache sets that are disabled. By moving the DSIndex we can control the fraction of cache sets that remain enabled, and thus the hit rate of the cache. BATMAN regulates the movement of DSIndex by dynamically monitoring the cache hit rate and comparing it to the target access rate (TAR, 80% in our case). We provision the system to monitor the hit rate of the cache and to increase and decrease DSIndex, as described below:

1) Structures:: BATMAN monitors the access rate of the DRAM cache using two 16-bit counters: *AccessCounterCache* and *AccessCounterTotal*. AccessCounterCache is used to track the total number of cache accesses (e.g. reads, writebacks, probes, etc.) while AccessCounterTotal is used to track the total number of accesses to the DRAM cache as well as the memory. When the AccessCounterTotal overflows, we halve (right shift by one) both counters.

2) Operation:: If the request maps to a set index beyond the DSIndex (Request-B in Figure 10), it is serviced in a normal manner – lookup the cache, if the line is found then return it, if not obtain the line from the memory and install it in the set. If the request maps the set index less than the DSIndex (Request-A in Figure 10), then the request directly goes to memory, without the need to lookup the cache.

3) Regulating DSIndex and Hysteresis:: BATMAN continuously monitors the observed service ratio (by computing the ratio of AccessCounterCache to AccessCounterTotal) to determine whether the DRAM cache accesses are exceeding the TAR. If the ratio exceeds the TAR, BATMAN increases the DSIndex until the ratio of the two counters is within a 2% guard-band of the desired TAR. If the observed ratio is lower than the TAR, BATMAN decreases DSIndex until the ratio of the two counters is within the 2% guard-band. Before increasing DSIndex, BATMAN first invalidates the set referenced by the DSIndex and ensure to write back dirty lines to the FM.[2]

4) Storage Overheads:: The proposed implementation of BATMAN requires a total storage overhead of only eight bytes: two 16-bit counters and a 32-bit DSIndex. BATMAN leverages the existing hardware for cache management, and requires minimal modifications to the cache access logic.

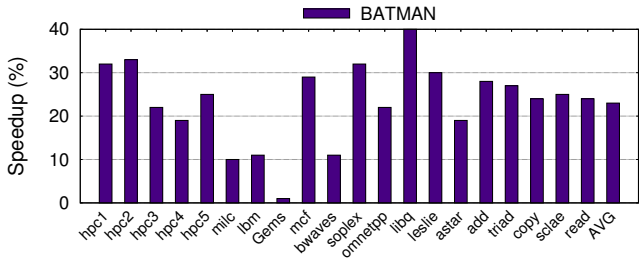

Fig. 11. Performance Improvement of BATMAN for DRAM cache.

C. Performance Improvement from BATMAN

Figure 11 shows the speedup from BATMAN. BATMAN improves performance across all workloads by an average by 23%. BATMAN for cache improves bandwidth in two ways: first, making the bandwidth of FM usable, and second is that misses to the disabled sets that would have been misses in the baseline too would now avoid the tag lookup bandwidth for the DRAM cache. This means, the overall bandwidth benefit from BATMAN exceeds more than just the bandwidth provided by the FM. This explains why the performance of several workloads exceeds 25%. For application whose working set is larger than NM, BATMAN is able to regulate the access rate by enabling all the cache sets, hence the performance improvement is small.

[2]We explain the case of contiguity in disabled sets for simplicity. If the frequently referenced pages happen to get located at sets that are far away from the DSIndex, it may take a long time for the DSIndex to reach such region of hot pages. The traversal time can be reduced by using *hashed* set disabling, whereby only every Nth set is eligible for disabling and DSindex gets changed by N every time. We use N=5 in our design.

D. Access Rate of Hardware Cache with BATMAN

Since BATMAN bypasses cache lookups for a fraction of memory accesses, instead of reporting the cache hit rate, we simply report the percentage of memory requests (from L3) that get satisfied by the NM cache (as opposed to the FM). Figure 12 shows the access rate for the NM for both our baseline system and BATMAN. In the baseline, the access rate of NM is on average 85%, with several workloads exceeding 90%. With BATMAN, the access rate reduces to the target access rate of 80%. On average, BATMAN has on average 79% access rate of NM, and thus our proposed design is quite effective at controlling the access rate of NM.

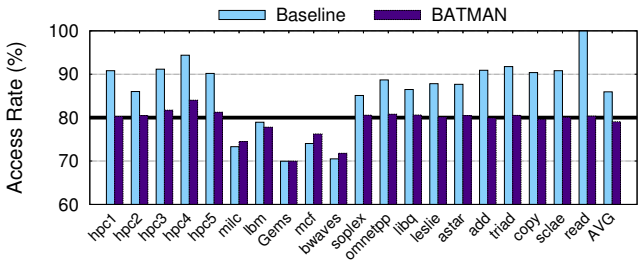

Fig. 12. Access Rate for the baseline and BATMAN.

VII. Results and Analysis

A. Bandwidth Improvement

As our goal is to maximize the bandwidth utilization in the system. We show the effectiveness of BATMAN at maximizing memory system bandwidth by explicitly measuring the bandwidth used in each configurations. We measure the memory bandwidth by monitoring the busy time of memory channel bus, and report the average bandwidth improvement for each benchmark suites. Figure 13 shows the bandwidth improvement by BATMAN in three configurations: a system without migration (No Migration), with page migration (Page Migration), and with cache line migration (Cache). We observe that on average, BATMAN improves memory bandwidth utilization by 11%, 10%, and 12%, respectively. The maximum bandwidth increase are consistently from STREAM benchmarks of 17-19%. Note that the improvement numbers are with respect to the baseline in each configuration.

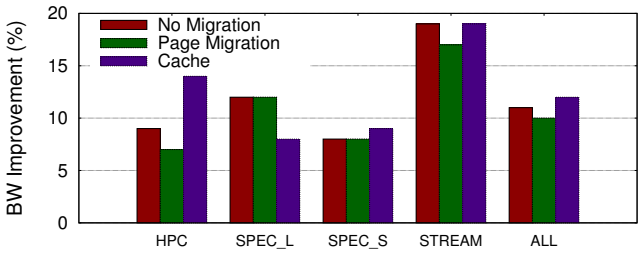

Fig. 13. System Bandwidth Improvement. No Migration, Page Migration and Cache are with respect to respective baselines

B. Sensitivity Study for Bandwidth Ratio

In our default system, we assume Near Memory has 4X bandwidth of Far Memory. We also conduct a sensitivity study to vary the ratio of Near and Far Memory bandwidth ratio from 2X to 8X. The results are shown in Figure 14. When Far Memory is half bandwidth of Near Memory, BATMAN is able to improve performance by more than 30%, because aggregate bandwidth is 1.5 times the Near Memory bandwidth, providing huge boost in available bandwidth. When the ratio is 8X, using Far Memory can increase the total bandwidth by 12.5%, which translates to 5-9% performance improvement.

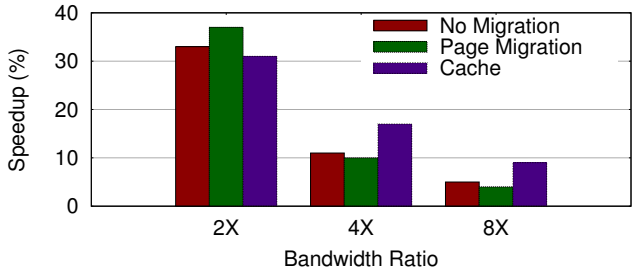

Fig. 14. Sensitivity Study of Different Bandwidth Ratio

C. Study Across A Larger Set of Workloads

We show detailed results for only 20 workloads due to space limitations; however we run our study on a much larger set of 35 workloads. We use s-curve to show performance improvement of all the 35 workloads used in our study in Figure 15. Note that all configurations are normalized to respective baseline configurations, and we sort the benchmarks by the value of the performance improvement. On average, BATMAN improves performance by 11% with maximum of 33% in a system without migration, 8% with maximum of 26% in a system with page migration, and 17% with maximum of 40% in a system configured with a cache. Note, that for the three sets of experiments with 35 workloads each (a total of 105 data points) we observe only one case of performance degradation (4% slowdown for *mcf* for the system with no data migration). Thus, our proposed scheme is not only high performing but is also robust across workloads and systems.

VIII. Related Work

A. DRAM Cache

Several recent studies have looked into using high bandwidth DRAM as a cache. Most of the work try to improve cache hit rate [8, 6], and are not aware of the additional bandwidth provided by FM. Thus, these schemes do not maximize the overall system bandwidth.

One of the prior work that consider using Far Memory bandwidth is Mostly-Clean DRAM cache [10], which uses a miss predictor to access to Far Memory in parallel with the cache, in order to mitigate the high cache latency in case the cache has significant amount of contention. However, the parallel access to both NM and FM increases memory

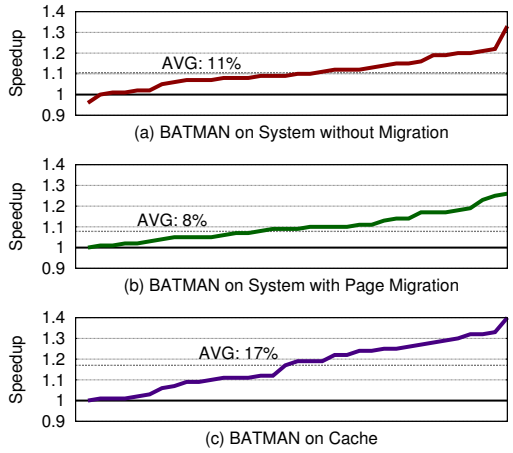

Fig. 15. Performance Improvement of BATMAN over a larger set of 35 workloads for the 3 systems. Note the performance shown on the S-curve is normalized with respect to the respective baseline.

traffic, and consumes even more bandwidth (two accesses per request). Hence, the aggregate memory system bandwidth does not increase. In contrast, BATMAN tries to use FM bandwidth in addition to the NM bandwidth and thus, does not increase the overall memory traffic.

B. Cache Optimization

A significant amount of research effort has been devoted to cache optimization. At the level of on-die cache, traditional practice is to improve hit rate, since high hit rate is typically correlated to improving performance. Therefore, most of the traditional approaches were trying to improve cache hit rate [25, 26, 27, 28, 29, 30]. However, our proposal deoptimizes cache hit rate at the level of DRAM cache or hybrid memory system in order to maximize the total bandwidth in the memory system. We show that when the constraints are different, it may make sense to reduce cache hit rate if cache hit rate is higher than required.

C. Software Optimization

Several page migration policies have been proposed in the last decades to improve the performance of Non-Uniform Memory Architecture (NUMA), where the local nodes is a order of magnitude faster in terms of latency than the remote nodes [17, 31, 32, 33]. The effort mainly focuses on how to improve the system performance in a shared memory model. For example, reducing the ping-pong effect in the shared memory architecture, which two nodes are accessing the same page [32].

Other software optimization including cache-blocking algorithm [34], user-level thread scheduling in NUMA environment [35] are proposed to optimized for latency reduction. However, in the case of hybrid memory system with high-bandwidth memory which have similar latency compared to FM, we show that these proposals may need to be revisited.

IX. Summary

Emerging memory technologies such as HBM, HMC, and WIO provide 4-8x higher bandwidth than traditional memory at similar random access latency. These technologies are used in tiered-memory systems where *Near Memory (NM)* is typically composed of low-capacity high bandwidth memory and *Far Memory (FM)* is typically composed of high-capacity low bandwidth traditional memory. Tiered-memory systems focus on servicing the majority of requests from NM by migrating data between NM and FM. We show that this approach wastes FM bandwidth especially when the application working set fits into NM. In such situations, performance can be improved by splitting accesses the between NM and FM.

This paper shows that splitting the accesses between NM and FM is directly proportional to the relative bandwidth offered by the respective memory systems. We propose *Bandwidth Aware Tiered-Memory Management (BATMAN)* that dynamically optimizes the target access rate in three different types of systems (a) a system that configures NM as OS-visible memory and statically maps pages between NM and FM (b) a system that configures NM as OS-visible but dynamically migrates pages between NM and FM (c) a system that configures NM as a hardware-managed DRAM cache. We demonstrate that the principles of BATMAN are applicable to all three systems, and our proposed implementation is both simple and highly effective. Our studies on a 16-core system with 4GB of NM and 32GB of FM show that BATMAN improves performance by as much as 40% and by 10-22% on average depending on the configurations, while requiring less than twelve bytes of hardware overhead.

As new memory technologies emerge, and a new set of constraints develop, we show that rethinking the management of the memory hierarchy can yield significant benefit. While we evaluate BATMAN for a hybrid memory system consisting of only stacked DRAM and commodity DRAM, the insights and solutions are applicable to other hybrid memory systems too. Exploring such extensions is a part of our future work.

References

[1] *HMC Specification 1.0*, 2013. [Online]. Available: http://www.hybridmemorycube.org
[2] JEDEC, *High Bandwidth Memory (HBM) DRAM (JESD235)*, JEDEC, 2013.
[3] Micron, *HMC Gen2*, Micron, 2013.
[4] *1Gb_DDR3_SDRAM.pdf - Rev. I 02/10 EN*, Micron, 2010.
[5] *DDR4 SPEC (JESD79-4)*, JEDEC, 2013.
[6] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 404–415. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485957
[7] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian, "CHOP: Adaptive filter-based dram caching for CMP server platforms," in *HPCA-16*, 2010.
[8] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked dram caches," in *Micro-45*, 2011.

[9] 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 *MICRO-45*, 2012.

[10] J. Sim, G. H. Loh, H. Kim, M. O'Connorand, and M. Thottethodi, "A mostly-clean dram cache for effective hit speculation and self-balancing dispatch," in *MICRO '12*, 2012.

[11] L. Zhao, R. Iyer, R. Illikkal, and D. Newell, "Exploring DRAM cache architectures for CMP server platforms," in *ICCD-25*, 2007.

[12] F. Bellosa, "When physical is not real enough," in *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, 2004.

[13] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, "Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support," in *Supercomputing*, 2010.

[14] M. Ekman and P. Stenstrom, "A case for multi-level main memory," in *Proceedings of the 3rd workshop on Memory performance issues*, ser. WMPI '04, 2004.

[15] H. Huang, P. Pillai, and K. G. Shin, "Design and implementation of power-aware virtual memory," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2003.

[16] G. H. Loh, N. Jayasena, J. Chung, S. K. Reinhardt, J. M. OConnor, and K. McGrath, "Challenges in heterogeneous die-stacked and off-chip memory systems," in *3rd Workshop on SoCs, Heterogeneous Architectures and Workloads*, 2012.

[17] M. A. Holliday, "Reference history, page size, and migration daemons in local/remote architectures," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS III. New York, NY, USA: ACM, 1989, pp. 104–112. [Online]. Available: http://doi.acm.org/10.1145/70082.68192

[18] J. D. McCalpin. Stream: Sustainable memory bandwidth in high performance computer. [Online]. Available: http://www.cs.virginia.edu/stream/

[19] "Intel core i7 processor specification." [Online]. Available: http://www.intel.com/processor/corei7/specifications.html

[20] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. H. Pugsley, A. N. Udipi, A. Shafiee, K. Sudan, and M. Awasthi, *USIMM*, University of Utah, 2012.

[21] D. Kaseridis, J. Stuecheli, and L. K. John, "Minimalist open-page: A dram page-mode scheduling policy for the many-core era," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 24–35. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155624

[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 *PLDI*, 2005.

[23] E. Perelman, G. Hamerly, M. V. Biesbrouck, and B. C. Timothy Sherwood, "Using simpoint for accurate and efficient simulation," in *ACM SIGMETRICS Performance Evaluation Review*, 2003.

[24] S. Ross, *A First Course in Probability*.

[25] S. M. Khan, D. A. Jimnez, and D. B. B. Falsafi, "Using dead blocks as a virtual victim cache," in *PACT-19*, 2010.

[26] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: ACM, 2010, pp. 60–71. [Online]. Available: http://doi.acm.org/10.1145/1815961.1815971

[27] D. A. Jiménez, "Insertion and promotion for tree-based pseudolru last-level caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 284–296. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540733

[28] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry, "The evicted-address filter: A unified mechanism to address both cache pollution and thrashing," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 355–366. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370868

[29] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 381–391. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250709

[30] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. Washington, DC, USA: IEEE Computer Society, 2012, pp. 389–400. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2012.43

[31] R. P. Larowe, Jr. and C. Schlatter Ellis, "Experimental comparison of memory management policies for numa multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 4, pp. 319–363, Nov. 1991. [Online]. Available: http://doi.acm.org/10.1145/118544.118546

[32] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox, "Numa policies and their relation to memory architecture," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IV. New York, NY, USA: ACM, 1991, pp. 212–221. [Online]. Available: http://doi.acm.org/10.1145/106972.106994

[33] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum, "Scheduling and page migration for multiprocessor compute servers," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VI. New York, NY, USA: ACM, 1994, pp. 12–24. [Online]. Available: http://doi.acm.org/10.1145/195473.195485

[34] G. Golub and C. Van Loan, *Matrix Computations*, ser. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 1996. [Online]. Available: http://books.google.com/books?id=mlOa7wPX6OYC

[35] S. Blagodurov and A. Fedorova, "User-level scheduling on numa multicore systems under linux," in *in Proc. of Linux Symposium*, 2011.