Energy-Efficient In-Memory Data Stores on Hybrid Memory Hierarchies

Ahmad Hassan
HANA Cloud Computing, SAP
ahmad.hassan@sap.com

Hans Vandierendonck
Queen's University of Belfast
h.vandierendonck@qub.ac.uk

Dimitrios S. Nikolopoulos
Queen's University of Belfast
d.nikolopoulos@qub.ac.uk

ABSTRACT

Increasingly large amounts of data are stored in main memory of data center servers. However, DRAM-based memory is an important consumer of energy and is unlikely to scale in the future. Various byte-addressable non-volatile memory (NVM) technologies promise high density and near-zero static energy, however they suffer from increased latency and increased dynamic energy consumption.

This paper proposes to leverage a hybrid memory architecture, consisting of both DRAM and NVM, by novel, application-level data management policies that decide to place data on DRAM vs. NVM. We analyze modern column-oriented and key-value data stores and demonstrate the feasibility of application-level data management. Cycle-accurate simulation confirms that our methodology reduces the energy with least performance degradation as compared to the current state-of-the-art hardware or OS approaches. Moreover, we utilize our techniques to apportion DRAM and NVM memory sizes for these workloads.

1. INTRODUCTION

Power and energy efficiency have become major concerns for the design of data centers. Main memory is a key energy consumer but high memory capacities are increasingly used to provide high throughput services. For example, Facebook stores 75 % of its non-image data in main memory [13] in order to avoid I/O bottlenecks. Increasing the capacity of main memory is an attractive option to achieve high throughput, however, DRAM technology has hit scaling and power barriers [13, 28]. The ability of DRAM technology to scale below 40 nm feature sizes is yet to be confirmed [28, 13]. With significant leakage power and high refresh power, DRAM-based main memory consumes 30-40 % of the total server power [1, 20, 15]. DRAM size directly impacts the power consumption of servers, which reduces its attractiveness.

Non-Volatile Memory (NVM) is emerging as a compelling main memory technology due to high density and low leakage power. Various novel NVM technologies have been proposed over recent years, each with different strengths and weaknesses in energy, performance, durability, density and scalability, and each with different likelihoods of making it to mass production. The main contenders are Phase Change Memory (PCM), Spin Transfer Torque memory (STT-RAM) and Memristors [12]. Resistive RAM (RRAM) memory is a memristor technology [12]. These technologies are byte addressable and exhibit zero refresh power [28, 22, 16]. However, NVM has several weaknesses. Reading and writing often takes longer than for DRAM and consumes more energy, with writing suffering more than reading [14, 28]. There is an asymmetry in read and write cost (e.g., PCM read and write latency is approximately 4.4× and 12× times DRAM latency). Similarly, dynamic energy of PCM read and write is approximately 2× and 43× times of DRAM respectively [14, 28]. Consequently, NVM is not a drop-in replacement for DRAM.

Hybrid memory combines a DRAM part and a NVM part. With careful data placement, hybrid memory can exhibit the latency and dynamic energy of DRAM in the common case, while rarely exposing the latency and high dynamic energy of NVM. Capacity is increased by NVM without expending the static energy of DRAM. Hybrid memory requires data management policies implemented at some level of the system stack. Most studies propose such policies implemented in hardware memory controllers [14, 28, 22], or at the operating system level [8, 23, 25, 6]. Contrary to prior research, the work presented here focuses on managing hybrid memory systems at the *application level*. We specifically show that the energy of the system can be reduced by carefully controlling the placement of objects on a hybrid DRAM-RRAM memory system, at the application level. The goal of this paper is to demonstrate that hybrid main memory systems can provide performance close to that of a DRAM-only memory system and consume energy close to that of an NVM-only memory system. We pursue this goal using application-level management of hybrid memory because we believe that such custom solutions pay off tremendously at the scale of data centers. The practical effect of our work is a reduction of the total cost of ownership of data centers as well as an increase of the feasible main memory size in servers.

Our approach starts with application-level object profiling that identifies all data objects in an application, i.e., global variables and heap- and stack-allocated data. The profiler measures the frequency and type of memory accesses to each object. The profiler enables us to estimate the latency, dynamic energy and static energy consumed by memory accesses incurred for each individual object in the application.

We apply the profiler to analyze two data-intensive applications, an analytics-oriented database and an in-memory key-value store (Section 3). We find that overall few objects critically affect latency. This is key to the success of our work, as placing those objects on DRAM avoids the higher latency of NVM.

Next we decide the placement of objects on DRAM vs. NVM by evaluating first-order performance and energy models on the metrics collected during profiling (Section 4). Moreover, the methodology allows us to calculate appropriate sizes for DRAM and NVM for these workloads.

Our final contribution is to apply our methodology (Section 5) and to demonstrate that application-level, profile-guided object placement in hybrid memory can reduce memory system energy by about 80.02 % (Table 4). The corresponding hybrid memory system design consists of large NVM and small DRAM, which is used as an inclusive cache.

Our profiling tool and data placement methodology have been previously described [11], where they were applied to CPU-intensive applications. This work applies these ideas to data-intensive workloads that are relevant for modern data centers. Moreover, we present a programming API (subsection 4.3) and argue that such an API is feasible to implement in modern OSes. We present an API and hybrid memory placement policies section 4 for two important classes of applications: in-memory column stores and key-value stores (section 3). We demonstrate that in both classes of application, static energy of main memory is the major portion of total energy and main memory accesses show strong preference towards a small fraction of the objects. Thus, few objects are critical to performance and demand placement on DRAM. Reducing the static energy consumption of the infrequently accessed objects is paramount. We propose static data placement techniques to minimize AMAE in hybrid systems with minimal impact on AMAT (section 5). We further classify objects as DRAM or NVM friendly for both classes of application.

2. BACKGROUND AND RELATED WORK

NVM technology is promising to replace or augment DRAM in the main memory system. However, there are significant variations in performance and power between NVM technologies. For example, PCM consumes 2.1× and 43.1× the DRAM energy for a read and a write operation respectively, while STT-RAM consumes less energy than PCM [18]. Similar variations hold for latency in various NVM technologies. Because of NVM's attractive scaling and power properties, several works consider combining DRAM and NVM [8, 14, 22]. In these hybrid memory systems, it is crucial to place data on the most appropriate type of memory.

Recent research has investigated how to manage hybrid memory by the hardware, OS, or the application. *OS-level* memory management must identify main memory access properties of the application at the granularity of virtual memory pages. Migrating virtual memory pages may lead to sub-optimal placement decisions when objects with different access frequencies are placed on the same page. The main benefit of OS-level page placement is that it is transparent to the application. Dhiman *et al* [8] demonstrated 30 % energy reduction by modifying the OS paging policy for hybrid memory. Their approach uses page access frequency to

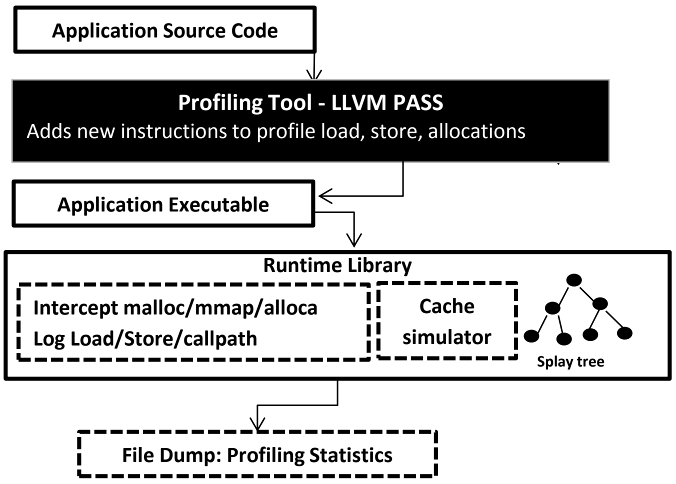
Application Source Code
Profiling Tool - LLVM PASS
Adds new instructions to profile load, store, allocations
Application Executable
Runtime Library
Intercept malloc/mmap/alloca
Log Load/Store/callpath
Cache simulator
Splay tree
File Dump: Profiling Statistics

Figure 1: Object analysis tool

trigger page migration. Similarly, Ramos *et al* [23] use page access frequency, which is collected from a customized memory controller. Ramos *et al* [23] use multi-level queues for ranking OS pages. This process requires significant number of cache misses in order to rank pages within the queues and decide the correct placement of pages on hybrid memory.

At the *hardware level*, prior research explored hybrid memory management via modifications to the organization and scheduling policies of the memory controller. Yoon *et al* [27] propose row-buffer locality aware caching policies for hybrid memory systems. They store rows with frequent misses on DRAM whereas rows with good locality are stored on NVM. This approach not only improves latency but also the lifetime of NVM. Lee *et al* [14] use DRAM as a buffer to hide the latency of PCM. They only write dirty cache blocks to PCM and buffer multiple writes to the same location. They show improvements in both performance and energy. Qureshi *et al* [22] propose a write cancellation policy to improve the read latency of PCM. Read accesses remain pending while slower writes are in progress on the same bank. By canceling writes and re-doing them after reads finish, performance increases by 3× and PCM lifespan increases from 3 to 9.7 years.

Application-level data placement requires programmer intervention and distinct solutions per application. However, the solutions can be optimized to suit a particular application. Various studies [31, 34, 32, 33, 17, 36] have explored data management on NVM at the application level. Li *et al* [17] explore the use of NVM in scientific applications by estimating what application objects to store in NVM. They apply a heuristic based on measured read/write ratios, memory reference rates, and sizes of all global and heap objects. They show that 31 % of the application working set is suitable for NVM and demonstrate power savings up to 27 %, compared to a DRAM-only system. The heuristic used by Li *et al* identifies almost exclusively read-only data as suitable for storage on NVM. Our work presented here disputes this conclusion. Jorge et al. [35] proposed software level techniques for persisting user level data structures. They proposed Software Persistence Memory (SoftPM) abstractions that define persistence containers in the user level code. The main strength of this technique is the ease of use and min-

Table 1: Latency (ns) and dynamic energy (nJ) for a 64-byte access and leakage power (mW/GB).

	Latency	Dyn. Energy	Leakage
	R/W, [ns]	R/W, [nJ]	[mW/GB]
DRAM	30.5/30.5 [21]	11.76/25.35 [21]	451 [21]
RRAM	64.98/95.41 [30]	13.33/31.44 [30]	4.23 [16]

imal code modification. Their results showed speed-up of 83 % for SQLite application. Mirhoseini et al. [36] proposed a user level technique for data encoding on PCM which reduces the PCM energy consumption. They worked on the assumption that PCM set and reset have different energy cost. Their technique reduces the energy cost by implementing the data encoding which used PCM bit-wise manipulation ability such that only the modified bits are overwritten. This not only improves the performance but also reduce the unnecessary write operations. Their techniques minimize memory energy consumption up to 44 %. Boncz et al. [4, 3] proposed the random access database join operators optimized for main memory bottleneck. They implemented a partitioned hash join operator that addresses the memory bottleneck problem and makes efficient use of caches on modern hardware.
In this work, we evaluate the suitability of storing both read and write-dominated objects on NVM using first-order analytical models of performance and energy. Moreover, we propose and evaluate actual data management policies for in-memory databases and provide a method to select the capacity of DRAM and NVM.

3. WORKLOAD CHARACTERIZATION

We developed a tool to identify all data *objects* in a software system and measure for each of these objects whether the object is best placed on DRAM or on NVM. We first describe our methodology and then apply it to two classes of application that are widely used in today's data centers: an in-memory database and an in-memory key-value store.

3.1 Analysis Methodology

In our methodology, an object is a piece of memory that is indivisible for the purpose of data placement and migration in hybrid memory hierarchies. Typically, objects map one-to-one to program variables (global variables and stack-allocated variables) as well as heap allocations.

We developed a run-time profiling library to register the creation, destruction and access of each object in the application. The profiling library simulates an on-chip cache hierarchy such that we can estimate the number of reads and writes to main memory to the object. We use LLVM [19] to insert calls to the library in the analyzed workloads. The overall flow is depicted in Figure 1. Our tool is based on the work of Rul *et al* [24]. Further details on the tool are provided in [11].

The analysis provides for each object its size, lifetime, number of main memory reads and writes, number of level-1 cache reads and writes, call site of the memory allocation, as well as a number of other metrics. Using this information, we can utilize first-order memory energy models to estimate whether an object should be placed in DRAM or in NVM (see Section 4.1).

Table 2: Workload and Input Sizes

App.	Benchmark	Size	Data	Run
MonetDB [3]	TPC-H [26]	5 GB	5 GB	completion
Memcached [10]	Twitter [9]	20 GB	16M keys	24M GET + 1.2M SET
Memcached [10]	YCSB [7]	20 GB	8M keys	8M GET + 4K SET

3.2 In-Memory Database

MonetDB is a state-of-the-art in-memory, column-oriented database. We execute the TPC decision support (TPC-H) benchmark which models a complex decision support system, on a 5 GB database.

We instrumented MonetDB using our tool and collected profiling statistics for all 22 TPC-H queries. Figure 2 shows workload characterization results for a selection of queries that represent all the diverse behaviors of MonetDB. Each chart shows the cumulative distribution function (CDF) of a per-object statistic. Objects are assigned unique numeric IDs in the order of decreasing number of load/store accesses from MonetDB to the object.

The left graph shows the CDF of memory accesses made to each object (load/store instructions executed by MonetDB). The graph is trimmed to the top 300 accessed objects out of total 198 K objects. These graphs show very strong locality of accesses: nearly all loads and stores are directed to less than 0.2% of the objects.

The middle graph of Figure 2 shows the number of off-chip memory accesses to each object. Objects targeted frequently by load/stores instructions are expected to incur many off-chip memory accesses, but oftentimes this is not the case. This can be deduced from the non-convexity of the curves. Consequently, data management policies must focus on the objects that are most frequently accessed in off-chip memory.

Finally, the graph on the right of Figure 2 shows the cumulative sizes of the objects. The largely flat lines demonstrate that a small number of objects (about 78) take up most of the memory space and the remaining objects are relatively small. These 78 objects include both column and intermediate objects. Moreover, we observe that the large objects (which are either columns or intermediates) are either hot (left-most spike in the right graph) or cold (right-most spike). There is also a smaller spike near the left in the curves, which corresponds to fairly large objects that often require off-chip memory accesses. By analyzing the source code, we identified that these objects store intermediate data computed by the queries.

These observations lead to following placement strategy:

- **Columns:** Data columns of the database should initially be placed on NVM as this type of memory provides ample capacity. Whenever columns become accessed frequently, they may be copied to DRAM. As OLAP workloads do not modify the bulk of the database, it is preferable to replicate data on DRAM such that it does not need to be copied back. We found that only the small set of columnar data is accessed during the query execution. For example, for TPCH Q# 11 on scale factor (SF) 5, the maximum size of used column was 30.6 MB.

- **Intermediate data:** MonetDB generates many small objects (between 1 KB and 64 MB for SF 5) that store intermediate data sets as a result of running primitive operations on columns and other intermediate objects.

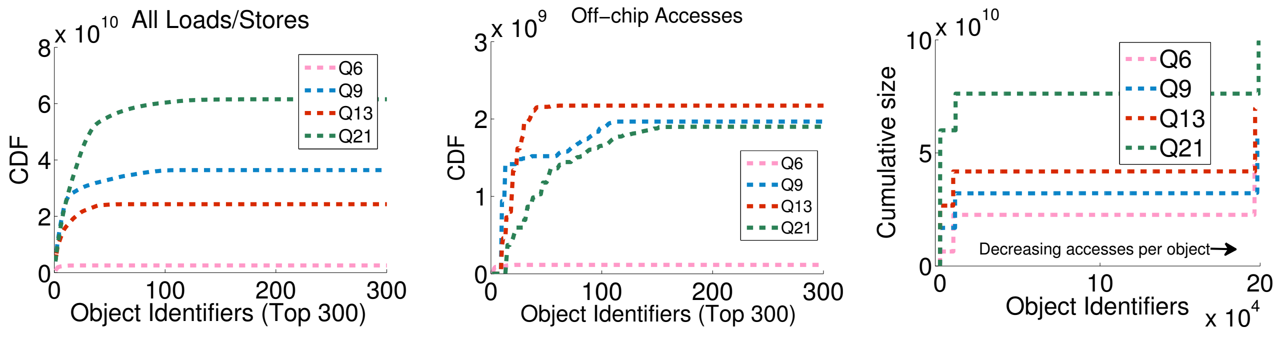

Figure 2: Workload Characterization of MonetDB. Left: cumulative distribution function (CDF) memory accesses for each object. Middle: CDF of number of off-chip accesses for each object. Right: CDF of the size of objects.

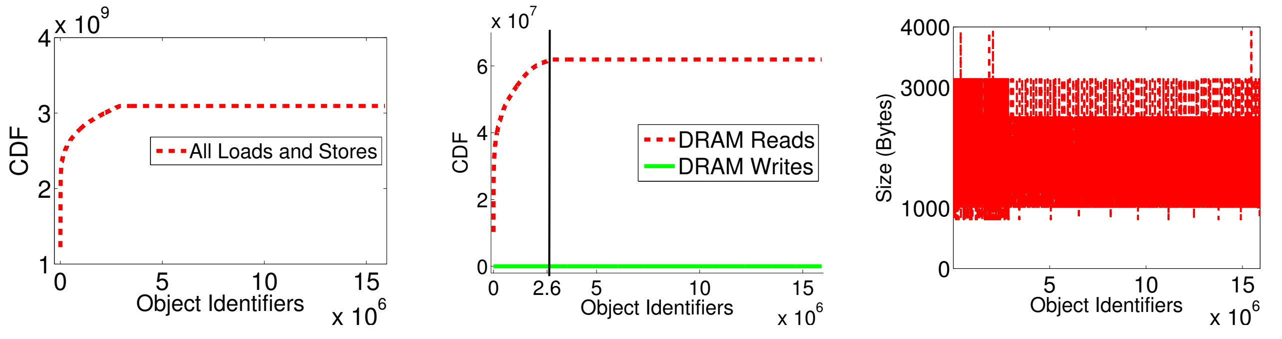

Figure 3: Workload characterization of memcached executing the Twitter workload.

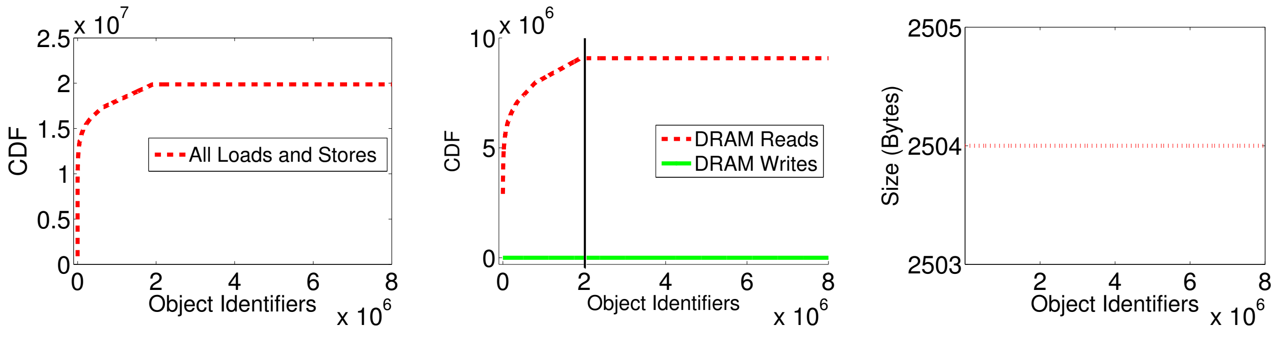

Figure 4: Workload characterization of memcached executing the YCSB workload.

For TPCH Q# 11, we measured that maximum working set size of primitive MAL operation is just 46.8 MB for SF 5. Moreover, apart from few exceptions, all the intermediates have 100 % of their accesses in the operator where they were created and never used again in subsequent operators. This materialization of intermediate results is performed with high locality [3], which is confirmed by our analysis. However, some intermediates generate a significant number of main memory accesses, so we distribute intermediate objects between DRAM and NVM.

- **Global and stack variables:** These variables take very little space in memory compared to the remainder of the application objects. Moreover, they generally have good locality. As such, we store them on NVM.

3.3 In-Memory Key-Value Store

Our second workload is memcached [10], an in-memory key-value store. Memcached implements a cache for key-value pairs and answers GET and SET requests on key-value pairs. For any request, memcached first consults a hash table indexed by the key to check if it holds the requested pair. If so, the hash table points to a chunk, a piece of memory large enough to store the key-value pair. Memcached allocates chunks on slabs, which are pre-allocated arrays of fixed-size chunks. By default memcached allocates slabs over 34 distinct chunk sizes. Chunks are allocated on a slab with the smallest chunk size that can hold the requested key-value pair (only on SET requests). Chunks are evicted using a first-in first-out (FIFO) policy.

We run memcached with two workloads, a Twitter workload [9] and the Yahoo Cloud Serving Benchmark (YCSB) [7]. The workload configurations are summarized in Table 2.

We have declared memcached's internal memory allocation interface for chunks to our workload characterization tool such that it can drill down its analysis to the granularity of chunks. As such, slabs do not show up in our analysis.

The Twitter workload is warmed up using 16 M keys. Then, we analyze the benchmark during 24 M GET requests assuming a 95 % GET-to-SET ratio. This workload results in a 95% memcached key hit rate. The YCSB workload simulates a scenario of Photo tagging with 95% read operations and 5% tag updates. We set up the workload with 8 M keys and execute the benchmark for 8 M GET requests (Table 2).

Figures 3 and 4 show the CDF of on-chip and off-chip access per object for Twitter and YCSB respectively. Both workloads show strong locality of reference. The hash table (not included in the graphs) and the chunks are the only important data structures in memcached. Based on these results, we devise the following data management policy:

- **Hash table:** The hash table is accessed in every GET and SET operation and is the hottest object in the application. In our runs it grows up to 275 MB and incurs regular cache misses. As such, it is most efficient to store it on DRAM.
- **Chunks:** The chunks make up the majority of memcached's data. Access patterns to chunks depend on the GET and SET commands received by memcached. As such, chunks are migrated between DRAM and NVM depending on user requests. For simplicity, we trigger data migration between DRAM and NVM using memcached's chunk replacement policy.
- **Remaining objects:** The remaining objects bear little relevance on performance, energy consumption and memory footprint. We conservatively store them on NVM as they mostly hit the cache.

4. HYBRID MEMORY MANAGEMENT

Our method is to first analyze the execution of applications and collect memory access statistics for individual objects. Then we evaluate a first-order analytical model of performance and energy to identify the most suitable memory technology to store that data. Finally, we design data placement policies based on this information.

4.1 Performance and Energy Model

The performance and energy models use the application characteristics and technology parameters for deciding the placement of objects on DRAM or NVM. We use first-order performance and energy models that focus on access latency [29]. Bandwidth issues can occur only when objects are frequently accessed, in which case we already select to place the object on DRAM due to the latency benefits.

The average memory access time (AMAT) incurred by memory accesses to one object o stored in a memory of technology τ is given by $AMAT_\tau(o)$:

$$AMAT_\tau(o) = \mu_r(o)L_{\tau,r} + \mu_w(o)L_{\tau,w} + (1-\mu_r(o))L_{LLC} \quad (1)$$

where τ is either DRAM or NVM, $L_{\tau,r}$ is the latency to read a cache block in memory (see Table 1) and L_{LLC} is the latency to access the last-level cache (LLC). $\mu_r(o)$ is the number of read memory accesses made to o per load or store operation. $\mu_w(o)$ is the of number write-backs made for o per load or store operation and $L_{\tau,w}$ is the latency to write a cache block to main memory. Note that we are assuming a single-level cache, i.e., an inclusive last-level cache, but the formula can be extended to multi-level caches.

Energy for an object consists of static energy, which is always present throughout the lifetime of an object and includes leakage and refresh energy, and dynamic energy, which is proportional to the frequency of memory accesses. Energy consumed by an object o is presented by the $AMAE_\tau(o)$ metric:

$$AMAE_\tau(o) = \mu_r(o)E_{\tau,r} + \mu_w(o)E_\tau + S(o)\ P_\tau T(o) \quad (2)$$

Here, $E_{\tau,r}$ and $E_{\tau,w}$ are the energy for reading and writing, respectively, a cache block to or from memory type τ. These parameters are computed from Table 1. The parameters $\mu_r(o)$ and $\mu_w(o)$ represent the read access and write accesses to memory, respectively, as in the definition of AMAT. P_τ is the average leakage power per byte for memory type τ. The parameters $S(o)$ and $T(o)$ represent the size and lifetime, respectively, of the object o.

4.2 Object Placement on Hybrid Memory

The goal of hybrid memory system is to minimize the static energy of DRAM (indicated by its size) and use DRAM effectively to hide the higher latency and dynamic energy of NVM. In Section 3 we performed an ad-hoc analysis to understand what objects exist in our applications and how they are best placed. The following outlines a quantitative approach based on the AMAT and AMAE models that place objects based on estimated performance and energy impact. The placement decided by this quantitative approach matches our ad-hoc analysis.

The algorithm uses the profiling information (in particular main memory reads and writes per object). For any object o, we calculate the metric $\Delta AMAT(o) = AMAT_{DRAM}(o) - AMAT_{NVM}(o)$ to estimate the potential speed-up by placing the object on DRAM. Similarly, we calculate $\Delta AMAE(o) = AMAE_{DRAM}(o) - AMAE_{NVM}(o)$ to estimate the energy gain by placing the object on NVM. The latter is typically a function of the trade-off between static and dynamic energy for the object.

We place objects such that energy is minimized and latency is raised by no more than a fixed percentage over a DRAM-only system. To this end, we sort the objects in order of increasing $\Delta AMAT(o)$ and place objects on DRAM in this order until DRAM is fully occupied. We partition the list of sorted objects $o_i, 1 \leq i \leq N$ by splitting the list at index s, such that objects $o_i, i \leq s$ are placed on DRAM and objects $o_i, i > s$ are placed on NVM. We determine the index s in order to meet the expected overall slowdown compared to a DRAM-only memory system:

$$\sum_{i=s+1}^{N} \Delta AMAT(o_s) \leq \lambda \sum_{i=1}^{N} AMAT_{dram}(o_s) \quad (3)$$

where λ is a user-configurable parameter. We set λ to 5% in this paper. The objects at the front of the sorted list are most sensitive to memory latency. These objects take priority for placement on DRAM. We placed objects one-by-one, starting at the front of the list, and estimate AMAT and AMAE if all objects considered so far are placed on DRAM and the others on NVM. The resulting curves (Figure 5) allow us to determine a good size for DRAM and NVM memory.

Table 3: MonetDB: Comparison of an NVM-only system (NVM), software placement (SWP) and RaPP with DRAM-only system.

	CPI Increase (%)			Energy Savings (%)		
Query	NVM	SWP	RaPP	NVM	SWP	RaPP
2	25.32	1.21	21.23	96.02	92.46	93.73
6	43.21	2.82	42.50	94.14	92.93	93.03
9	34.18	3.20	32.13	91.44	89.46	90.43
13	37.13	3.14	34.71	92.91	88.22	89.57
14	28.57	2.17	21.73	96.29	91.98	93.27
17	24.85	1.48	16.35	96.66	95.10	94.12
18	46.66	3.81	44.31	89.44	86.22	88.92
21	38.35	3.10	35.19	93.13	90.32	92.89

In hybrid system, the application programmers choose where to allocate memory via the memory allocation libraries. In our work, the OS and the underlying hardware memory controllers does not influence the memory page placement on one memory or another. We suggest the application-level techniques for data placement. The application programmers decide themselves where to place data on the basis of application knowledge and workload characteristics.

4.3 Hybrid Memory API

The hybrid memory API allows programmers to encode that data placement as suggested by the profiling data and the models. We control data placement by splitting the virtual address range in two portions, one each corresponding to DRAM and NVM. This is a feasible solution as 48-bit virtual address space, on 64-bit OS, is large enough. Moreover, in x86 systems, the BIOS passes the physical memory map to the OS through the E820 controller. By extending the memory map with distinct entries for the DRAM and NVM portions of memory, the OS can match those physical memories with physical address ranges using similar data structures as required to enforce NUMA partitions. Moreover, the OS can match physical address ranges (corresponding to DRAM or NVM) to the corresponding virtual address ranges using those data structures as used to enforce NUMA partitions.

Data placement is controlled through memory allocation. We propose a lightweight extension to the mmap() system call where a flag is added to allocate memory on either DRAM or NVM. This choice is extended to the user-level allocation library, where the `malloc` calls are extended. Moreover, in our implementation, we ensure that `free` can operate without programmer-supplied information on the memory type. This reduces programming effort.

5. EXPERIMENTAL EVALUATION

We used the state-of-the-art cycle accurate GEM5 [2] simulator for the validation of data placement in MonetDB. We built the static executable of MonetDB using LLVM and simulated MonetDB on GEM5 in system emulation mode. We implemented custom memory allocator for DRAM and NVM. We extended Doug Lea's memory allocator (dlmalloc)[1], version 2.8.6, to build a custom memory allocator, as it easily allows to create two distinct heaps within the same application using MMAP. The custom allocator use distinct virtual memory address (VMA) regions for each memory type using the MAP_Fixed flag to `mmap` to control the starting address of each region. We implemented NVMalloc and DRAMalloc interface to allocate memory from VMA-I or VMA-II respectively.

[1]ftp://gee.cs.oswego.edu/pub/misc/malloc.c

Table 4: MonetDB: Percentage of energy savings through optimized data placement.

Q	%E	Q	%E	Q	%E	Q	%E
1	69.49	7	69.68	13	70.44	19	79.77
2	79.74	8	68.50	14	69.44	20	69.87
3	62.24	9	65.86	15	69.42	21	60.24
4	71.45	10	68.51	16	79.34	22	80.02
5	68.12	11	79.85	17	77.94		
6	70.50	12	74.95	18	70.72		

We implemented application-level data management policies in memcached and MonetDB as outlined in Section 3. We modified Memcached memory model with a DRAM slab of fixed chunk size which has a FIFO queue. The SET allocates memory on NVM and the object is migrated to DRAM on first access. We modified MonetDB to allocate columns and hot intermediate objects on DRAM. The rest of the objects are placed on NVM.

5.1 Evaluation of Data Placement

Figure 5 summarizes the performance vs. energy trade-off achieved with data placement in hybrid memory. It shows the average latency per memory access (AMAT) as more objects are moved from NVM onto DRAM, according to our quantitative placement methodology (Section 4).

For MonetDB (Figure 5, left), we observe for several queries that placing all objects on NVM (the left-most point on the curve) incurs high AMAT (between 60 and 80 CPU cycles per access) but memory access energy is lowest. On the other extreme, a DRAM-only system yields lowest AMAT but incurs high energy per memory access. By moving frequently accessed objects to DRAM one by one, we observe that AMAT quickly becomes equal to the DRAM-only AMAT, while requiring that only a fraction (around 20 %) of the data is placed on DRAM. Table 4 summarizes the energy savings archived in a hybrid memory comprises of DRAM and RRAM.

Similar results are observed for memcached (Figure 5), although in this case we use dynamic migration of data. Again we observe that storing around 18 % of Twitter or 23 % of YCSB data on DRAM yields near-optimal performance. In this case it must be noted though that different degrees of locality in the request stream may impact this decision.

5.2 Simulation-Based Validation

We validate our methodology using GEM5 simulator and compare our approach with the state-of-the-art RaPP policy [23]. We extended the GEM5 system emulation with missing Linux system calls that were necessary to execute MonetDB on GEM5 such as getdents, munmap and readdir. We configured GEM5 with two memory controllers, one interfacing with 512 MB DRAM and one interfacing with 8 GB RRAM. The timing parameters for RRAM are derived from the Micron DDR3 datasheet [21] and prior literature [14, 16]. Specifically, we derived the tRCD, tRP, tRRDpre and tRRDact parameters for RRAM following Lee *et al* [14]. We calculated tRCD as 2× larger than DRAM, tRP as

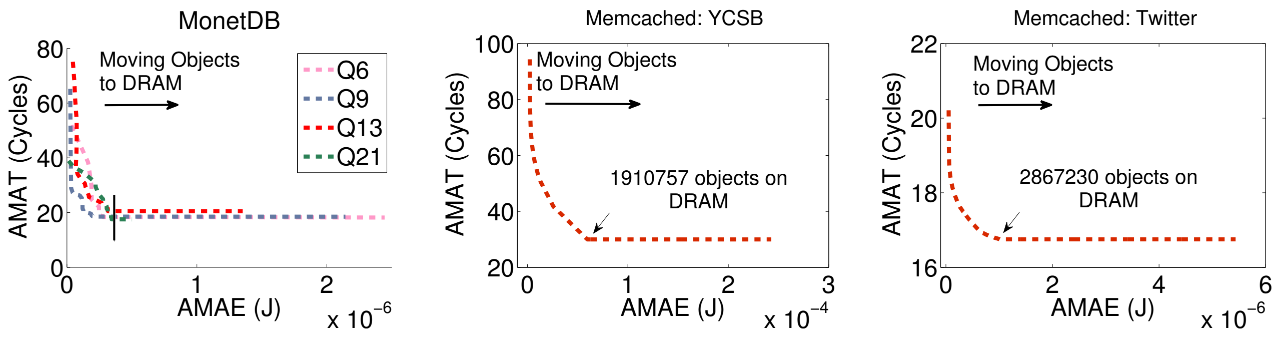

Figure 5: AMAT versus AMAE

2.2× larger than DRAM, tRDDpre is 6 ns and tRDDact is 6.5 ns. Moreover, we modified the GEM5 memory controller to avoid write-back of clean cache blocks to memory. We fast-forwarded the first 100 M instructions and ran the simulation with 30% sampling rate until the completion of queries. We also implemented state-of-the-art Rank-based Page Placement (RaPP) [23] in GEM5. In RaPP, we used 15 queues and DRAM expiry time of 100μs. For RaPP, we configured 128 MB DRAM and 8 GB RRAM.

Software object placement out-performs RaPP on performance (CPI) for all the MonetDB queries. We present the results of four representative queries in Table 3. These results show that with our approach CPI never increases by more than 3.81 %, whereas with RaPP the CPI increases up to 44.31 %. In fact, RaPP does not perform much better than an NVM-only memory system. RaPP does not perform well for MonetDB because RaPP requires learning time prior to migrating pages between NVM and DRAM. This learning process works fine if the pages are accessed frequently but MonetDB doesn't access pages too frequently due to good cache locality.

Please note that the estimated energy savings for object placement (Table 4) are less than those measured in the simulator (Table 3). This is due to the simulation setup: memory size is constant in the simulator (8.5 GB), while the profiling tool predicts energy savings relative to the actual size of the memory footprint. As such, energy savings predicted with in GEM5 simulator are invariably high if the workload leaves a major part of memory unutilized.

5.3 Dynamic Data Placement

We have shown in section 5 that our proposed methodology works well for static data placement. We know that only a small set of objects are hot in main memory where as many objects remain cold during the execution of any MAL operation in MonetDB. Our analysis confirms that, even within a single BAT, only 1 or 2 heap objects are hot and others are cold. We are extending this work to devise operator-level heuristics for dynamic data placement and migration on hybrid memory. Such heuristics will take into account the operator type, estimated size of result, type and size of input.

6. CONCLUSION

Non-volatile memory is crucial in order to sustain growing in-memory data sets for data intensive computing. We have investigated the feasibility of orchestrating data placement and migration for hybrid memory at the application level. For in-memory data stores, we have demonstrated that pages are the wrong granularity of data for data management on hybrid memory. Using application-level data management techniques, we found a sweet spot where energy of the system is significantly reduced while maintaining the performance close to DRAM-only system. We found that it is surprisingly easy to design and implement such management at the application level for an analytical database and an in-memory data store.

Using hybrid memory and application-level data placement, we can find a sweet spot where the energy of main-memory system is significantly reduced while keeping the performance degradation up to 3.81 %.

For future work, we intend to elaborate on the implementation of the hybrid memory API in the OS. Moreover, we will consider hardware support for data migration at the hardware level, e.g., through DMA controllers and selective cache flushing.

7. ACKNOWLEDGEMENT

We thank the anonymous reviewers for their valuable feedback. This work is supported by the European Community's Seventh Framework Programme (FP7/2007-2013) under the NovoSoft project (Marie Curie Actions, grant agreement 327744) and the NanoStreams project (grant agreement 610509).

8. REFERENCES

[1] L. A. Barroso and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 4(1):1–108, 2009.

[2] N. Binkert, *et al.* The GEM5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[3] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, Dec. 2008.

[4] P. A. Boncz, S. Manegold and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access, VLDB 1999

[5] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, June 1997.

[6] J.-H. Choi *et al.* OPAMP: Evaluation framework for

optimal page allocation of hybrid main memory architecture. In *ICPADS*, pages 620–627, 2012.

[7] B. F. Cooper *et al.* Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[8] G. Dhiman, R. Ayoub, and T. Rosing. PDRAM: A hybrid PRAM and DRAM main memory system. In *DAC*, pages 664–469, 2009.

[9] M. Ferdman *et al.* Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *ASPLOS*, pages 37–48, 2012.

[10] B. Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5, Aug. 2004.

[11] A. Hassan, H. Vandierendonck, and D. S. Nikolopoulos, Software-Managed Energy-Efficient Hybrid DRAM/NVM Main Memory. In *Computing Frontiers*, *to appear*, 2015.

[12] Y. Ho, G. M. Huang, and P. Li. Nonvolatile memristor memory: Device characteristics and design implications. In *DAC*, pages 485–490, 2009.

[13] ITRS. *International Technology Roadmap for Semiconductors*, 2011.

[14] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *ISCA*, pages 2–13, 2009.

[15] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. Keller. Energy management for commercial servers. *Computer*, 36(12):39–48, 2003.

[16] E. Doller, "Forging a future in memory - new technologies, new markets, new applications," in *Hot Chips Tutorials*, 2010.

[17] D. Li *et al.* Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications. In *IPDPS*, pages 945–956, 2012.

[18] E. Kultursay, M. Kandemir, A. Sivasubramaniam, O. Mutlu. Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative In *ISPASS*, 2013.

[19] The LLVM compiler infrastructure. http://llvm.org.

[20] D. Meisner, B. T. Gold, and T. F. Wenisch. Powernap: eliminating server idle power. In *ASPLOS*, pages 205–216, 2009.

[21] Micron TN-41-01: Calculating Memory System Power http://www.micron.com/products/support/power-calc

[22] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *ISCA*, pages 24–33, 2009.

[23] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page placement in hybrid memory systems. In *ICS*, pages 85–95, 2011.

[24] S. Rul, H. Vandierendonck, and K. De Bosschere. A profile-based tool for finding pipeline parallelism in sequential programs. *Parallel Comput.*, 36(9):531–551, Sept. 2010.

[25] D.-J. Shin *et al.* Adaptive page grouping for energy efficiency in hybrid PRAM-DRAM main memory. In *RACS*, pages 395–402, 2012.

[26] Transaction processing performance council. http://www.tpc.org.

[27] H. Yoon. Row buffer locality aware caching policies for hybrid memories. In *Intl. Conf. on Computer Design (ICCD)*, pages 337–344, 2012.

[28] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *ISCA*, pages 14–23, 2009.

[29] H. Vandierendonck, A. Hassan, and D. Nikolopoulos. On the energy-efficiency of byte-addressable non-volatile memory. *Comput. Archit. Letters*, PP(99):1–1, 2014.

[30] X. Dong, *et al.* NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *TCAD*, 31(7):994–1007, July 2012.

[31] S. Chen and Q. Jin. Persistent B+-trees in Non-volatile Main Memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.

[32] S. D. Viglas. Write-limited Sorts and Joins for Persistent Memory. *Proc. VLDB Endow.*, 7(5):413–424, Janary 2014.

[33] S. Pelley *et al.* Storage Management in the NVRAM Era. *Proc. VLDB Endow.*, 7(2):121–132, October 2013.

[34] S. Chen, P. B. Gibbons and S. Nath. Rethinking database algorithms for phase change memory. CIDR 2011

[35] J. Guerra *et al.* Software persistent memory Proceedings of the 2012 USENIX conference on Annual Technical Conference

[36] A. Mirhoseini, M. Potkonjak and F. Koushanfar. Coding-based Energy Minimization for Phase Change Memory Proceedings of the 49th Annual Design Automation Conference