CAMEO:A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache

Chiachen Chou† Aamer Jaleel‡ Moinuddin K. Qureshi†

†*School of Electrical and Computer Engineering*
Georgia Institute of Technology
{*cc.chou, moin*}*@ece.gatech.edu*

‡*VSSAD*
Intel Massachusetts Inc.
aamer.jaleel@intel.com

***Abstract*—This paper analyzes the trade-offs in architecting stacked DRAM either as part of main memory or as a hardware-managed cache. Using stacked DRAM as part of main memory increases the effective capacity, but obtaining high performance from such a system requires Operating System (OS) support to migrate data at a page-granularity. Using stacked DRAM as a hardware cache has the advantages of being transparent to the OS and perform data management at a line-granularity but suffers from reduced main memory capacity. This is because the stacked DRAM cache is not part of the memory address space. Ideally, we want the stacked DRAM to contribute towards capacity of main memory, and still maintain the hardware-based fine-granularity of a cache.**

We propose *CAMEO*, a hardware-based *CAche-like MEmory Organization* that not only makes stacked DRAM visible as part of the memory address space but also exploits data locality on a fine-grained basis. CAMEO retains recently accessed data lines in stacked DRAM and swaps out the victim line to off-chip memory. Since CAMEO can change the physical location of a line dynamically, we propose a low overhead *Line Location Table (LLT)* that tracks the physical location of all data lines. We also propose an accurate *Line Location Predictor (LLP)* to avoid the serialization of the LLT look-up and memory access. We evaluate a system that has 4GB stacked memory and 12GB off-chip memory. Using stacked DRAM as a cache improves performance by 50%, using as part of main memory improves performance by 33%, whereas CAMEO improves performance by 78%. Our proposed design is very close to an idealized memory system that uses the 4GB stacked DRAM as a hardware-managed cache and also increases the main memory capacity by an additional 4GB.

I. Introduction

Dynamic Random Access Memory (DRAM) technology is facing several critical challenges in bandwidth and scalability. The performance and scalability of main memory system can be improved by combining DRAM with alternative memory technologies that provide either low power (LPDRAM) [1], or high bandwidth (3D-DRAM) [2], or high density (Phase Change Memory) [3, 4]. Future memory systems are likely to consist of heterogeneous memory technologies. One of the key questions in architecting such heterogeneous memory systems is deciding the functional role (e.g. cache or main memory) for different technology components. In this paper, we consider the architectural choices for a system that integrates 3D die-stacked DRAM with commodity DRAM.

Stacked memory (also called as 3D-DRAM) is a revolutionary memory technology that stacks multiple layers of DRAM to provide a high-bandwidth low-latency memory structure. Recent prototypes have shown that die-stacked memory can be as large as several hundred megabytes to a few gigabytes, and can provide almost an order of magnitude of higher bandwidth compared to traditional DRAM [2, 5, 6, 7]. However, the capacity of stacked DRAM may not be sufficient to fully replace the off-chip commodity DRAM in a cost-effective manner. Thus, a practical way to use stacked DRAM is to design it in conjunction with commodity DRAM. Typically such a system would architect the stacked DRAM as either a hardware-managed cache [8, 9, 10, 11, 12, 13], or as part of the OS-visible main memory [14, 15, 16, 17, 18].

Architecting stacked DRAM as a cache has the advantage that it is transparent to the software and hence can be employed without relying on support from software vendors. Furthermore, the caching structure can be managed at a fine granularity of CPU cache lines (typically 64 bytes), which uses memory bandwidth efficiently. Recent proposals [10, 11, 12] have shown that stacked DRAM caches can significantly improve performance. These proposals work well when the capacity of the stacked DRAM is sufficiently small compared to the commodity DRAM.

As the technology for manufacturing stack memory matures, the size of stacked memory can ultimately account for a quarter or even half of the overall capacity of DRAM in the memory system. So, for future systems, using stacked DRAM only as a cache may become less attractive as the stacked DRAM would not contribute towards the OS visible main memory. As a result of the loss of memory capacity, applications with large memory footprints can suffer higher rate of page faults and thus suffer slowdown due to frequent access to storage.

An alternative usage is to architect stacked DRAM as part of the OS-visible main memory space. In such a memory system, pages resident in stacked DRAM are serviced with high bandwidth and low latency, while pages resident in commodity DRAM are serviced with high latency and low bandwidth. The performance of such a system can be improved if the OS moves data for locality and ensures that

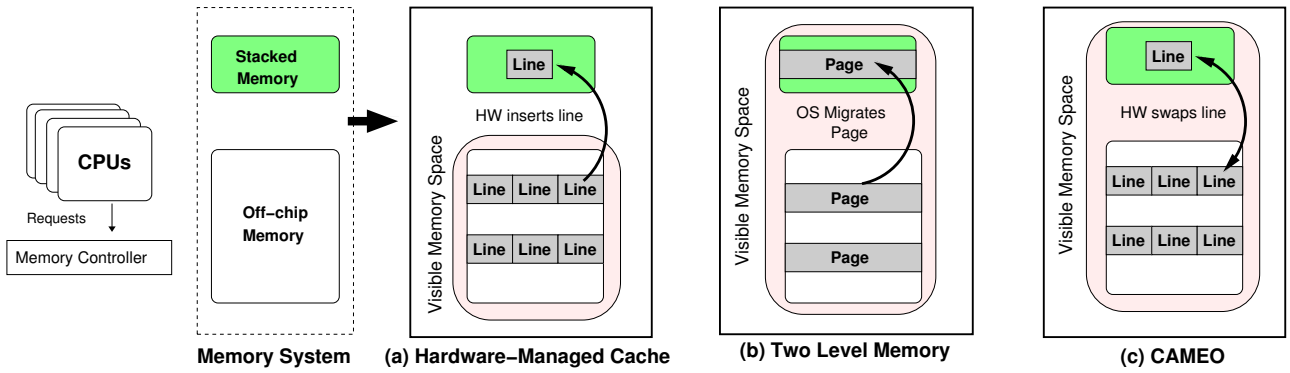

Figure 1. Architecture for using Stacked DRAM (a) Hardware-managed Cache at Line granularity (b) OS-managed Two Level Memory, at page granularity, and (C) Our proposal, CAMEO, with large memory capacity and transparent management at line-granularity.

frequently referenced pages are located in stacked DRAM. However, this requires OS support, and dictates that the granularity of data movement between stacked DRAM and commodity DRAM must be at a page granularity (4KB in our study). Unfortunately, not all lines within a given page are necessarily referenced, which causes inefficient use of memory system space and bandwidth. Therefore, moving data at page granularity can cause bandwidth bottlenecks.

Ideally, we want the size of memory space seen by the OS to be as large as possible, and we also want a cache-like system that is able to perform data migration in a fine-grained fashion. To this end, we propose a *CAche-like MEmory Organization (CAMEO)*. CAMEO is a hardware mechanism that makes both stacked memory and commodity memory form a continuous memory space as seen by the OS, yet leverages the locality on a cache line basis. CAMEO accomplishes this by retaining the recently accessed cache line in stacked memory. On a reference to a line in off-chip memory, CAMEO upgrades the requested line by swapping it with another line from stacked memory. This allows subsequent references to the upgraded line to be serviced with low latency and high bandwidth.

Figure 1 illustrates CAMEO and also compares it to other stacked memory designs.[1] The system being considered has two memory components: stacked memory and off-chip memory. Stacked memory can be either a cache or seen as a part of the memory space. Figure 1(a) shows the usage of stacked DRAM as cache. The cache is organized at a fine granularity of a cache line. However, stacked DRAM is not part of the memory address space. Figure 1(b) shows the usage of stacked DRAM as main memory. While this system has larger visible memory space, obtaining data locality requires OS support for page migration. Our goal is to provide a full memory capacity while retaining the properties of stacked-DRAM caches, as shown in Figure 1(c).

CAMEO relies on swapping of data lines to retain the benefits of stacked memory. Since swapping changes the physical location of a given memory line, CAMEO maintains a *Line Location Table (LLT)* to track the physical location of all memory lines. As each cache line in the system requires an entry in the LLT, the size of the LLT becomes quite large (several tens of megabytes). It is impractical to store such a large table on-chip in SRAM, or embed it within the L3 cache. Ideally, we want the LLT to have low overhead and low latency. To keep our design practical, we propose to co-locate the LLT entries with data lines in stacked DRAM. This design not only has the advantage of avoiding extra SRAM storage for maintaining LLT, but also of avoiding the serialization of LLT lookup for data lines that are resident in stacked DRAM.

Unfortunately, for data lines that are not resident in stacked memory, CAMEO suffers from high latency as memory access gets serialized with LLT lookup. To avoid this serialization, we propose a *Line Location Predictor (LLP)*. The LLP predicts the physical address of the cache line, and if the cache line is predicted to be in off-chip memory, CAMEO fetches the predicted location in parallel with the LLT access. The prediction is verified from the LLT entry, and if found correct, the data line is used. We observe that the history of the last location (based on instruction addresses) serves as a good predictor for the location of the line. Even With simple hardware structures (storage overheads of less than 1 KB per core), our LLP predicts the physical location of a line with 90% accuracy.

CAMEO does not require any software support or TLB changes for data migration and line tracking. Thus, CAMEO avoids the reliance on software vendors to deploy a memory system containing stacked DRAM.

[1] At the first glance, CAMEO may seem like an exclusive cache, but it is fundamentally different. Regardless of whether the cache is inclusive or exclusive, it always maintains a subset of main memory. Cache exclusion is a property of a "bigger cache" with respect to the "smaller cache", and not about main memory. For example, current AMD designs [19, 20] use exclusive L3, but it does not mean that those designs increase main memory capacity by the size of the L3 cache. It only means that the L3 does not hold lines that are in L2 (main memory is still inclusive of all caches). Designing a hardware cache that is exclusive of main memory is non-trivial as it would mean that every miss in the cache be handled by the OS (requiring remapping of pages), and will necessitate that data migration happens at page granularity. In essence, this will degenerate into TLM-Dynamic. To the best of our knowledge, no prior work has proposed a line granularity hardware-managed cache, which is exclusive of main memory.

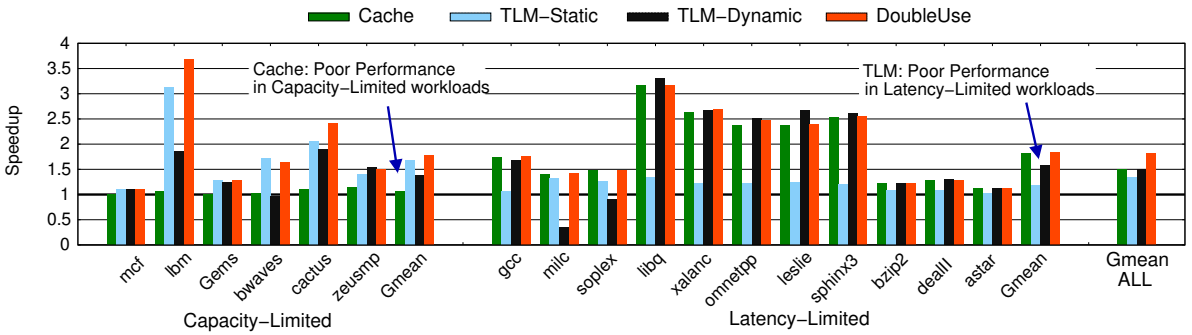

Figure 2. Performance evaluation of a system, where stacked DRAM is one quarter of total DRAM capacity, implemented as hardware cache, or Two-Level Memory (with and without page migration), or an idealistic "DoubleUse" system that uses stacked DRAM as a hardware cache and increases memory capacity by the size equivalent to the stacked memory.

We evaluate our system with 4GB stacked memory and 12GB of off-chip memory. Using stacked DRAM as cache improves performance by 50%, using stacked DRAM as main memory improves performance by 33%. CAMEO outperforms both designs by improving performance by 78%. We show that the performance of CAMEO is within 4% of an idealized system (termed *DoubleUse*) that uses the 4GB stacked DRAM as a cache and also increases the memory capacity by an additional 4GB.

II. Background and Motivation

Recently the research community has started investigating different emerging technologies to integrate with DRAM for future memory systems. One such technology is stacked DRAM, which provides roughly half the latency of traditional DRAM at about 8x higher bandwidth [2, 10, 6, 11]. Unfortunately, the size of stacked DRAM modules might not be large enough to replace off-chip memory, although the capacity of stacked DRAM may be as much as quarter of overall capacity of DRAM in the system [5, 7], as shown in Figure 3. Hence, it is likely that stacked DRAM and commodity DRAM will co-exist in future systems, and existing proposals use the stacked DRAM as either a hardware managed cache or as part of main memory.

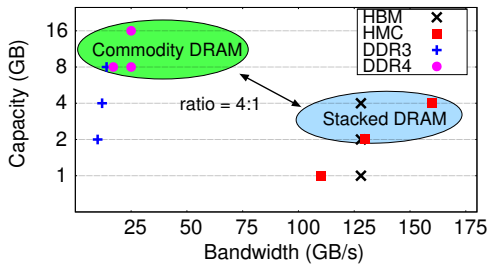

Figure 3. DRAM Capacity and Bandwidth (Data collected from various specifications [2, 5, 7, 6, 21], Y-axis in logscale).

A. Using Stacked DRAM as Cache

Using stacked-DRAM as a cache has the advantage that it can be deployed without relying on OS support. Such a cache is used as an intermediate cache between the last level cache (LLC) of the processor and main memory. The key challenges in architecting DRAM caches include designing a low-overhead low-latency tag store, and avoiding the serial lookup latency for accessing memory. Recent work [10, 18, 11, 12, 8] has addressed these challenges by integrating tag store within the DRAM array and using cache hit predictors [11, 12] to avoid tag look-up serialization latency. In this work, we will evaluate the performance impact of using stacked DRAM as hardware cache using the state-of-the-art Alloy Cache [11]. Alloy Cache optimizes the cache structure for overall system performance by minimizing hit latency with a direct-mapped cache and co-locating the tag store entry with the data line.

Figure 2 shows the performance improvement of architecting stacked memory, which is one quarter of total DRAM capacity, as a DRAM cache for our baseline system (experimental methodology is in Section III). We classify the workloads into two categories: Capacity-Limited (memory footprint more than off-chip memory) and Latency-Limited (memory footprint fits in off-chip memory). Overall, using stacked DRAM as a hardware cache provides 50% improvement. However, for Capacity-Limited workloads the improvement is marginal, because using stacked memory as a cache makes it invisible to the OS.

B. Using Stacked DRAM as Two Level Memory

Using stacked DRAM as part of memory address space has the advantage of providing an effective capacity that is the sum of both stacked DRAM and commodity DRAM. We refer to such a system as *Two Level Memory (TLM)*. With TLM, the stacked memory space is made visible to the OS, thereby increasing the total memory capacity of the system. This allows the system to accommodate a larger number of pages and reduce the number of page faults, thus reducing the slowdown that happens due to accesses to storage.

The advantage of TLM is that it avoids the tag store overhead of a cache, as it leverages the existing paging mechanism to decide the physical location of the page. A simple way to deploy TLM is to statically partition the address space into low-latency region (stacked DRAM) and high-latency region (off-chip memory). Thus, TLM services

memory requests with either low-latency or high-latency depending on the physical location of the page address.

A straight-forward implementation of TLM is oblivious to the latency characteristics of different memories and randomly maps the pages across the memory address space. We refer to this as *TLM-Static*. Figure 2 shows the performance of architecting stacked memory as TLM-Static. As the total memory capacity visible to the OS gets increased, the Capacity-Limited workloads see significant benefits (67% on average). However, when workloads are not limited by memory capacity, the benefits are much reduced compared to hardware cache (18% vs.82%). This is because, cache retains the data lines with high locality in the stacked memory, whereas TLM-Static does not optimize for data locality.

C. Problem of Coarse-Granularity Migration

The Operating System can optimize TLM for data locality by migrating pages with high locality from off-chip memory to stacked memory. We refer to this configuration as *TLM-Dynamic*. TLM-Dynamic retains recently accessed pages in stacked memory. It does so by swapping a page that gets accessed in off-chip memory with a victim page in stacked memory. Unfortunately, such data migration must occur at a page granularity (4KB in our study). The cost of migrating data at page granularity is significantly high, as it entails a swap operation of 4KB between stacked memory and off-chip memory. Both memory modules must read and write the respective 4KB pages (a total memory activity of 16KB). When a significant fraction of lines are not referenced, such page granularity transfers become highly inefficient in terms of memory bandwidth.

Figure 2 shows the performance of architecting 4GB stacked memory as TLM-Dynamic. For Latency-Limited workloads, TLM-Dynamic has some performance degradation compared to Cache. However, for Capacity-Limited workloads, the overhead of data migration far outweighs the potential benefits. Overall, TLM-Dynamic has better performance than TLM-Static (50% versus 33%). Although data migration can optimize for locality, doing so at large granularity may limit the performance.[2]

D. Goal: Optimizing for Both Capacity and Locality

Ideally, we want an architecture that does not rely on OS support, provides full memory capacity, and still optimizes data locality at a fine granularity. To illustrate the performance of such a design, we evaluate an "idealistic" configuration, called *DoubleUse*, which not only uses stacked memory as a hardware cache but also increases the capacity of off-chip memory by the size of stacked memory. In essence, this is a theoretical configuration to show the potential improvement possible with having increased memory capacity and performing fine-grained data migration.

Figure 2 shows the performance of the DoubleUse system. For Latency-Limited workloads, DoubleUse performs similar to hardware cache, as these workloads do not need higher memory capacity. However, for Capacity-Limited workloads DoubleUse performs significantly better than hardware cache, and marginally better than TLM.[3] Overall, the DoubleUse system has a performance of 82%, whereas optimizing the system only for capacity (TLM-Static) provides 33%, optimizing for both capacity and locality at page granularity (TLM-Dynamic) provides 50%, and hardware cache provides 50%. The goal of our paper is to develop an organization that has cache-like properties of managing data at fine granularity, while still providing full memory capacity, and without relying on OS support. We discuss our experimental methodology before describing our solution.

III. Experimental Methodology

A. System Configuration

We use a Pin-based [22] x86 simulator with a detailed memory system model. Table I shows the configuration used in our study. The parameters for the L3 cache, and DRAM (both off-chip and stacked) are similar to the recent studies on stacked DRAM [10, 11]. For architecting stacked DRAM as a hardware cache, we use the Alloy Cache [11]. We consider a system where stacked DRAM accounts for 25% of the total DRAM capacity, hence we use 12GB off-chip memory provisioned with 4GB of stacked DRAM. We model a virtual to physical translation. The victim page is selected using a clock algorithm (if an invalid page is not found after probing five random locations). Page faults in our system are assumed to be serviced by a solid-state disk with a latency of 32 microsecond (10^5 cycles).

B. Workloads

We use a representative slice of 20-billion instructions from SPECCPU 2006 suite. The evaluation is performed by executing benchmarks in rate mode, where all cores execute the same benchmark. Given that our study is about the memory system, workloads that spend a negligible amount of time in memory are not meaningful for our studies. To capture the memory system activity for different applications, we classify the benchmarks based on memory working set and Miss Per Thousand Instructions (MPKI) in L3 cache. As our baseline system has 12GB memory, we label benchmarks that have a working-set size larger than

[2]Prior work [18] on using OS-based page remapping of frequently accessed pages from off-chip DRAM to stacked-DRAM had similar conclusions that when the overheads are modeled, the performance improvement of page migration reduces significantly. We analyze different page migration policies for TLM in Section VI-D.

[3]For libquantum, TLM-Dynamic performs better than DoubleUse. This happens because the stacked DRAM in DoubleUse is a direct-mapped cache and suffers from conflict misses, even if the working set would have fit in the stacked DRAM. With TLM-Dynamic, the OS based page allocation eventually moves all pages into stacked DRAM, hence TLM-Dynamic outperforms Double Use. However, on average, DoubleUse significantly outperforms TLM-Dynamic.

Table I
BASELINE SYSTEM CONFIGURATION

Processors	
Number of Cores	32
Frequency	3.2GHz
Core Width	2 wide out-of-order
Last Level Cache	
Shared L3 Cache	32MB, 16-way, 24 cycles
Stacked DRAM	
Bus Frequency	1.6GHz (DDR 3.2GHz)
Channels	16
Banks	16 Banks per rank
Bus Width	128 bits per channel
tCAS-tRCD-tRP-tRAS	9-9-9-36 bus cycles
Off-Chip DRAM	
Bus Frequency	800MHz (DDR 1.6GHz)
Channels	8
Banks	8 Banks per rank
Bus Width	64 bits per channel
tCAS-tRCD-tRP-tRAS	9-9-9-36 bus cycles
SSD Storage	
Page Fault Latency	32 micro seconds (100K cycles)

12GB as *Capacity-Limited*. For the remaining benchmarks (working set less than 12GB), we sort them based on Misses Per Thousand Instructions (MPKI), and group the benchmarks with MPKI greater than 1 as *Latency-Limited*.[4] Table II shows the L3 MPKI and memory footprint for the workloads used in our study. The virtual-to-physical mapping ensures that multiple benchmarks do not map to the same physical address.

Table II
WORKLOAD CHARACTERISTICS (32-COPIES IN RATE MODE)

Limited By	Name	L3 MPKI	Memory Footprint
Capacity	mcf	39.1	52.4GB
	lbm	28.9	12.8GB
	GemsFDTD	19.1	25.2GB
	bwaves	6.3	27.2GB
	cactusADM	4.9	12.8GB
	zeusmp	5.0	14.1GB
Latency	gcc	63.1	2.8GB
	milc	31.9	11.2GB
	soplex	28.9	7.6GB
	libquantum	25.4	1.0GB
	xalancbmk	23.7	4.4GB
	omnetpp	20.5	4.8GB
	leslie3d	15.8	2.4GB
	sphinx3	13.5	0.60GB
	bzip2	3.48	1.1GB
	dealII	2.33	0.88GB
	astar	1.81	0.12GB

C. Figure of Merit

We measure the execution time when all benchmarks in the workload finish execution (as benchmarks are run in rate mode the variation in completion time of different benchmarks within a workload is negligible). We report speedup of a given configuration as the ratio of the execution time of the baseline (with no stacked DRAM) to the execution time of that configuration.

[4]Due to space limitations, we do not show detailed results for the remaining SPEC benchmarks (working set $<$ 12GB and MPKI $<$ 1). For these workloads, cache, TLM, and CAMEO all have similar performance.

IV. CAMEO: ARCHITECTURE AND DESIGN

By integrating stacked DRAM with commodity-DRAM, we would like to provide a high-capacity and low-latency memory system. A Two Level Memory optimizes the first objective of high capacity, and a cache achieves the second objective using fine-grained organization and hardware management. To achieve both objectives simultaneously, we propose *CAche-like MEmory Organization (CAMEO)*.

A. Overview of CAMEO

Figure 4 provides an overview of CAMEO. In the example, the stacked memory has capacity of N lines, and the off-chip memory has capacity of 3N lines. Thus, combining stacked DRAM and off-chip memory would provide a visible address space of 4N lines. For simplicity, let us assume that the memory space starts from stacked memory and grows to the region of off-chip memory. To leverage locality like hardware cache, CAMEO keeps the recently accessed data line in stacked memory by swapping data lines between the two memory regions. We call the group of lines that can be mapped to a given location in stacked DRAM as a *Congruence Group*. For example, lines A, B, C, and D form a Congruence Group. We restrict that lines can only be swapped with another line from the same Congruence Group. This is similar to the group of lines contending for the same set in a hardware cache (to accommodate one line we evict another line from the same set). The number of Congruence Groups is equal to the number of lines in stacked memory. If there are N lines in stacked memory, then the bottom $log_2(N)$ bits of the requested line address identifies the Congruence Group of the line.

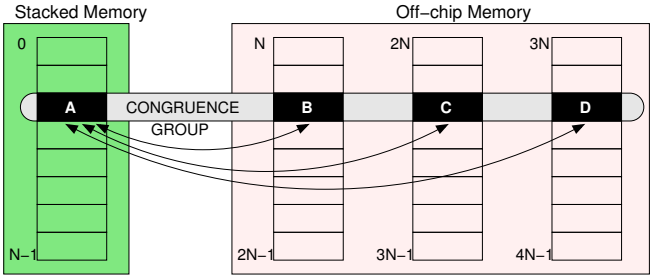

Figure 4. Lines A, B, C and D form a *Congruence Group*. CAMEO performs swapping only within the Congruence Group.

When a line in off-chip memory is accessed, say Line B, CAMEO would evict Line A from stacked memory and store Line B in the location of Line A. CAMEO would then store Line A in the off-chip memory where Line B was initially stored. Swapping maximizes effective capacity by ensuring that there is only one copy of the line in main memory.

Line Swapping: The swapping operation is performed in hardware using existing writeback and fill queues. As CAMEO operates on line granularity, a swapping operation is done as a writeback from stacked DRAM to main memory, and a demand read from main memory to stacked DRAM.

B. Line Location Table

CAMEO performs the swapping operation in hardware in a manner transparent to the Operating System. Such swapping causes a given line to relocate to another position within the Congruence Group. To correctly identify the position of a requested line, CAMEO must track the physical position of all data lines. The hardware structure that keeps track of this information is called the *Line Location Table (LLT)*. For each Congruence Group, the LLT keeps a record of the physical location of all the lines. We call the address requested by the LLC of the processor as the *Requested Address* and the real address where such line is located as *Physical Address*. As LLT is kept at the granularity of a Congruence Group, each LLT entry provides a mapping of all the lines in the Congruence Group. For example, for our configuration with 4GB stacked memory and 12GB off-chip memory, we would have four lines in the Congruence Group, and each entry in the LLT will be a four-entry tuple with two bits of location for each of the four lines.

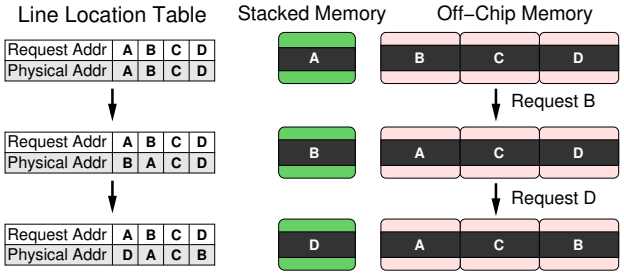

Figure 5. Operation of the Line Location Table (LLT), which keeps location information for each Congruence Group. Lines A, B, C, and D form a Congruence group, and operation of LLT is shown after two memory requests are performed.

Figure 5 illustrates the operation of LLT with an example. Lines A, B, C, and D belong to the same Congruence Group. Initially, the LLT entry contains an identity mapping where the physical addresses of the lines are identical to the real location. When a request to Line B is made, CAMEO would swap Line A and B, and record the new mapping in the LLT. When a subsequent request is made to say Line D, then CAMEO would swap Line B (which is in stacked memory) with Line D, and update the LLT entry accordingly. Thus, a line can move to any location within the Congruence Group (for example, Line B got moved within off-chip memory to the location of Line D). CAMEO uses the LLT much like the "tag-store" in a traditional cache to identify which line is resident in the stacked memory. However, unlike hardware cache, CAMEO also uses the LLT information to identify the real location of the line in the off-chip memory in case the line is not found in the stacked memory.

C. Design Challenges for the Line Location Table

To access a line with CAMEO, the request must first access the LLT to determine the physical location of the line. Only then can the memory controller decide whether the line should be obtained from the stacked memory or off-chip memory. To keep CAMEO practical, it is important that the storage and latency overheads of the LLT are kept to a minimum. However, this is a challenging task.

For our system with 4GB stacked memory and 12GB off-chip memory, each Congruence Group will have four lines. Thus, each LLT entry will be one byte (4 entries of 2 bits each). For a 16GB system, there would be 64 Million Congruence Groups (16GB divided by 256B, the size of the Congruence Group). Thus, the total size of the LLT for our system will be 64 MB. We discuss the design trade-offs and challenges in architecting an LLT of such a large size.

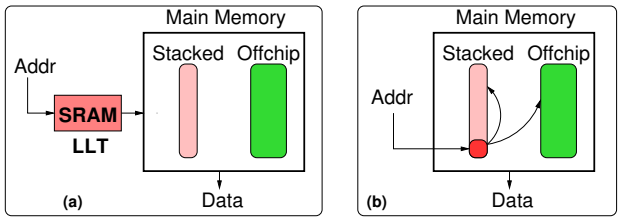

Figure 6. Options for LLT. (a) SRAM-based LLT incurs impractical storage overhead (b) LLT can be embedded in stacked DRAM but incurs indirection latency (first access for LLT, second for data).

1) SRAM-Based LLT (Impractical): The size of LLT (64MB) is greater than the size of the Last Level Cache (LLC) found in current microprocessors. Therefore, designing a LLT made of SRAM would incur unacceptably high overhead (in essence, sacrificing the L3 cache for storing LLT). Furthermore, accessing the LLT would still incur a latency overhead of as high as the L3 cache (24 cycles). Figure 6(a) shows the design of SRAM-based LLT. The requested line address probes the LLT, to identify the real location of the line, and access the main memory with this line address to obtain data. As this design incurs impractically high storage overhead, this design is only of theoretical importance and we do not consider it any further.

2) Embed LLT in Stacked DRAM (Practical but Slow): A more practical approach to design the LLT is to avoid the SRAM overheads by storing the LLT in the stacked DRAM. Figure 6(b) shows such a design that embeds the LLT in stacked DRAM. A portion of stacked DRAM is reserved to serve as the LLT. An incoming line address first indexes the LLT to obtain the LLT entry. Based on the real location of the line, then the second access is performed for obtaining data, either from the stacked DRAM or off-chip DRAM.

This approach, which we call *Embedded-LLT*, sacrifices some capacity of stacked memory for storing LLT, and makes this capacity invisible to the memory address space. Fortunately, the size of the LLT (64MB) is much smaller than the size of the stacked DRAM (4GB). We assume that the first 64MB of 4096 MB stacked memory is reserved for

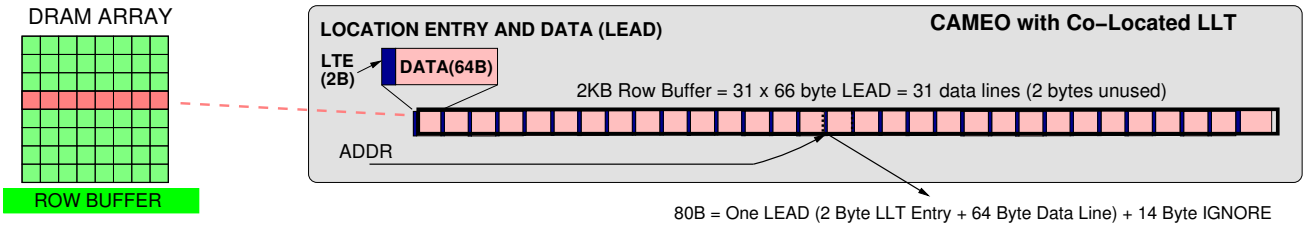

Figure 7. Organization of Co-Located LLT. The Location Table Entry (LTE) is co located with the data line to form a Location Entry and Data (LEAD) of 66 bytes. The 2KB row buffer stores 31 LEAD. Each access to stacked DRAM provides one LEAD.

the LLT, and the remaining 4032 MB is available to serve as main memory. Thus, with Embedded-LLT, 98.5% of the stacked memory is still available to serve as main memory.

Unfortunately, Embedded-LLT introduces the indirection latency of looking up the LLT before accessing data. This means, each request gets broken down into two requests, first to lookup the LLT (in stacked DRAM) and second to obtain data (either from stacked DRAM or off-chip DRAM). This latency overhead increases the effective latency of memory accesses and degrades performance.

D. Practical LLT by Co-location with Data Line

We expect that in the common case, the memory requests will be serviced by stacked memory. Therefore, if we can remove the serialization latency of LLT lookup for the lines that are resident in the stacked DRAM, then we can expect significant latency improvement over Embedded-LLT. We propose a design, called *Co-Located LLT*, which co-locates the LLT Entry with the data line. Each data line is appended with a Location Table Entry, to form an entity called *Location Entry and Data (LEAD)*. When the stacked DRAM is accessed, we get one LEAD. If the Location Table Entry in the LEAD identifies that the requested line is present in the stacked DRAM, we can directly use the data from the LEAD, without any extra access to the stacked DRAM. If the LEAD identifies that the line is in the off-chip memory, then a second access for desired location in off-chip memory is performed. Thus, Co-Located LLT can avoid the LLT lookup serialization for lines that are resident in the stacked memory.

The row buffer of the stacked DRAM used in our study is 2KB. Ideally, we would be able to accommodate 32 data lines (of 64 bytes each) in this row buffer. However, to implement Co-Located LLT, we must provision space at the granularity of a LEAD. We sacrifice memory space of one data line in the row buffer, and use it to support the Location Table Entry for the other 31 lines. Thus, each LEAD can have up to two bytes of Location Table Entry (we use one byte and keep one byte reserved for future use). The size of one LEAD is thus, 2+64=66 Bytes. The size of the data bus for the stacked DRAM used in our studies is 16 bytes, so we set a burst length of five, and obtain 80 bytes. We use the 66 bytes of LEAD and ignore the extra 14 bytes. Figure 7 shows the design of the Co-Located LLT for a given row buffer in stacked memory. The 2KB row buffer can accommodate 31 units of LEAD, resulting in a useful capacity of 31/32 (97%). For simplicity, we keep the assumption that the first 32MB in memory space in not visible to the operating system. Before accessing the stacked memory, we modify the physical address appropriately.[5]

E. Latency Comparisons of LLT Designs

An ideal design of LLT, termed *Ideal-LLT*, would incur zero overheads for LLT storage and latency. As soon as the memory controller receives the requested line address, it would know the real location of the line and would access the data from that location. Figure 8 compares the latency of Embedded-LLT and Co-Located LLT to Ideal-LLT. We assume that an access to stacked memory incurs 1 unit of latency and an access to off-chip memory incurs two units.

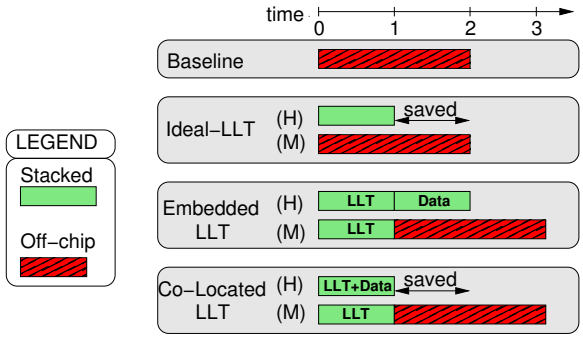

Figure 8. Access Latency Comparison for different LLT designs, for a system with stacked DRAM latency of 1 unit and off-chip DRAM latency of 2 units.

The analysis considers a single memory request serviced in isolation. For the baseline, the request is serviced by off-chip memory and would incur a latency of 2 units. With an Ideal-LLT, if the line is in stacked DRAM (case denoted as H), it will be serviced with a latency of one unit. However if the line is in off-chip memory (case denoted as M), it would be serviced with a latency of two units. For the Embedded-LLT, the LLT lookup takes one unit of time. Therefore, if the data line is in stacked DRAM, it would be serviced in two units (case H), and if the data is in off-chip memory, it would take 3 units (case M). Thus, Embedded-LLT has no latency

[5]For a given LineAddr X in stacked memory, the revised location of stacked memory is obtained using [(X + X/31)-LinesIn32MB]. Note, as we are dividing by a constant, the division operation can simply be performed with a few adders using residue arithmetic (31=32-1). This operation can be performed in parallel with the L3 access to hide the latency.

advantages for accessing data from stacked DRAM (albeit there may still be bandwidth benefits), and a slowdown for off-chip accesses. For the Co-Located LLT, if the data line is in stacked DRAM, it would be serviced in 1 unit (LLT access and data access happens in one transfer). If the data is in off-chip memory, then the lookup latency of LLT becomes serialized, and the total latency would be 3 units. Thus, Co-Located LLT has lower latency for data lines in the stacked DRAM. However, it has higher latency for off-chip accesses.

F. Performance Comparisons of LLT Designs

Figure 9 compares the speedup of CAMEO with Ideal-LLT, Embedded-LLT, and Co-Located LLT. As CAMEO provides a high memory capacity, there are benefits for capacity intensive workloads for almost all CAMEO configurations. As Embedded-LLT has high latency overheads, it results in performance slowdown for latency-sensitive workloads. Co-Located LLT, on the other hand, has lower latency when data lines are resident in stacked DRAM, hence the performance improvement is significant (on average 74%). However, there is still a significant performance gap between Co-Located LLT and Ideal-LLT (on average, 74% versus 80%). This is mainly because of the slowdown of off-chip accesses due to the LLT lookup. The next section describes solutions to avoid the serialization of LLT lookup.

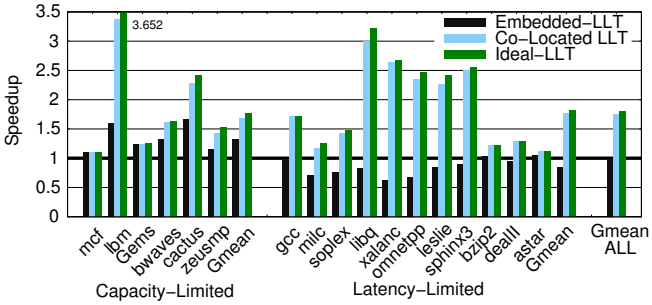

Figure 9. Speedup of different LLT designs. Embedded-LLT has high latency overheads, hence the slowdowns. Co-Located LLT has low latency for data lines in stacked DRAM, however because of higher off-chip latency the performance is lower than Ideal-LLT.

V. Memory Location Prediction

The Co-Located LLT avoids the latency of LLT lookup for lines resident in the stacked DRAM by fetching the LLT entry and data together. However, it still suffers from the latency of LLT lookup for lines that are resident in the off-chip memory. We propose a prediction mechanism that avoids the LLT serialization latency for the off-chip accesses. We first describe the framework for how such a predictor can be integrated in CAMEO, then the design of the predictor, followed by performance evaluation.

A. Avoiding LLT Latency with Location Prediction

Figure 10(a) shows the memory access with CAMEO. The off-chip access is serialized and happens only after accessing the stacked memory. We call such a model of memory access as *Serial Access Memory (SAM)*. Alternatively, we can predict the location of the line using a *Line Location Predictor (LLP)*. The organization of CAMEO with LLP is shown in Figure 10(b). If the LLP predicts that the location of the line is in off-chip memory, we can access both the stacked memory and off-chip memory in parallel. Only the predicted location in off-chip memory is accessed. If the line is found in stacked DRAM, then the prediction is ignored. However, if the line is not found in stacked DRAM, then the location provided by the LLP is verified with the LLT entry obtained from the stacked DRAM. If the prediction is correct, the line from off-chip location is used. Given that the off-chip access was made in parallel with stacked DRAM access, this scheme avoids the latency of LLT lookup.

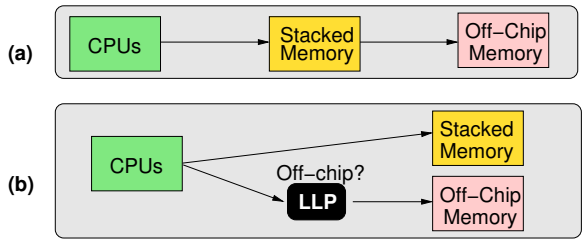

Figure 10. Avoiding LLT latency with prediction. (a) With SAM, off-chip access happens only after stacked-DRAM access (b) If access is predicted to be off-chip, the predicted location is accessed in parallel.

B. Line Location Predictor

If the LLP is accurate, we can avoid the serialization of LLT lookup for off-chip accesses in the common case. The main challenge for designing an effective LLP is that the LLP must decide upon the correct location from multiple candidate locations. This is unlike previous schemes on cache hit prediction [11, 12] that makes a binary decision between cache and memory. In our configuration, the line could be in any of the four locations, say 00, 01, 10, or 11, with location 00 being in stacked memory, and other three locations being in off-chip memory. Thus, the LLP must make a prediction out of four choices, as shown in Figure 11(a).

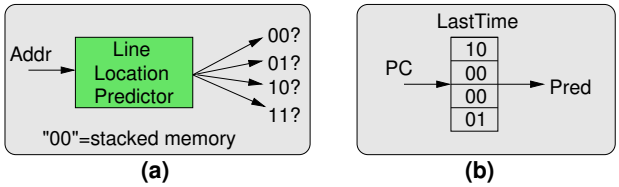

Figure 11. Line Location Predictor (a) LLP must make a 4-ary choice (b) A PC-based LLT implementation that predicts the location based on last-time.

To keep our predictor design simple, we exploit history in memory reference stream, as memory references are known to show good correlation with past behavior [23]. In particular, we make use of *Last Time Prediction*. We predict that the location table entry will provide the same location it provided the last time. A simple implementation of such a history-based last time predictor would be to keep a two-bit register called *Line Location Register (LLR)*, to track the physical address of the recent L3 miss. On the next L3 miss, if the location in LLR is in stacked memory (location 00), then serial access is used. Otherwise, the location provided by LLR is used to identify the off-chip location within the Congruence Group.

We can further enhance our predictor based on the observation that the memory reference stream tends to be heavily correlated with the instruction address that causes the memory access [24, 25] . Instead of a single LLR, we employ a table of LLRs, indexed by the instruction address of the L3 misses causing instruction. Our evaluation shows that using a 256-entry (8-bit index) table is quite effective at bridging the performance gap between serial access and perfect prediction. As each LLR is 2-bits, a table of LLR with 256 entries, would require 64 bytes. We employ a predictor on a per-core basis, so eight such prediction tables are employed, incurring a total storage overhead of 512 bytes. Thus, the Line Location Predictor in our design requires negligible overheads.

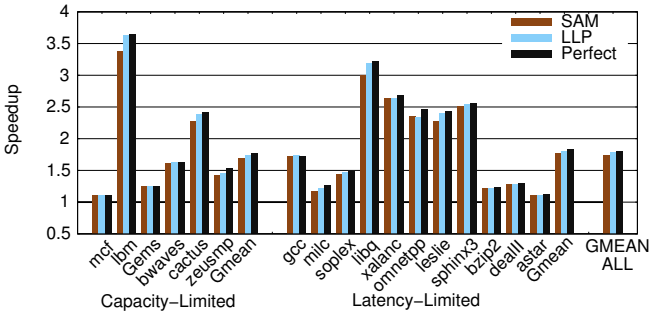

Figure 12. Speedup for no prediction, location prediction, and perfect prediction. On average, no prediction provides 68%, LLP provides 89%, and perfect prediction provides 94%.

C. Performance Results of LLP

Figure 12 shows the speedup for CAMEO using Co-Located LLT, with and without the LLP predictor. We also compare the performance with a prefect predictor that has 100% accuracy. On average, the performance improvement with SAM is 74%, and with perfect predictor is 80%. Our proposed Line Location Predictor provides an average performance of 78%, coming within 2% of a perfect predictor. Thus, even though our proposed implementation is simple and low overhead, it is still highly effective at obtaining most of the potential performance from location prediction.

D. Prediction Accuracy Analysis

For assessing the accuracy of LLP, we first describe five possible cases that can occur: 1) The physical location is in stacked memory, and the predictor predicts it as such. 2) The physical location is in stacked memory, but the predicted location is in off-chip memory. 3) The physical location is in off-chip memory, but the predictor gives a location in stacked memory. 4) The physical location is in off-chip memory, and the predicted location is correct and in off-chip memory. 5) The physical location is in off-chip memory, and the predicted location is not correct but still in off-chip memory. LLP makes accurate prediction in case 1 and 4, while case 2, 3, 5 are deemed as mis-prediction. Although case 2, 3, and 5 are mispredicted, they have different consequence in terms of latency and bandwidth. Case 2 wastes off-chip memory bandwidth, case 3 increases access latency, and case 5 is a combination of bandwidth waste and latency increase. Table III shows the percentage of each scenario for no prediction (serial access), prediction using LLP, and perfect predictor.

Table III
ACCURACY OF LINE LOCATION PREDICTOR

Serviced by	Prediction	SAM	LLP	Perfect
Stacked	Stacked	70.3	**68.4**	70.3
	Off-chip	0	1.8	0
Off-chip	Stacked	29.7	1.7	0
	Off-chip (OK)	0	**23.3**	29.7
	Off-chip (Wrong)	0	4.8	0
Overall Accuracy		70.3	91.7	100

SAM has 70.3% accuracy, which means higher latency for 29.7% accesses. LLP has an accuracy of 92%. For 7% of the accesses LLP causes useless memory access (case 2 and 5), and for 7% of the accesses it causes high latency (case 3 and case 5). However, for the 92% requests LLP is accurate, and provides both low latency and avoids wasteful bandwidth from parallel access. For the rest of the paper, we will assume CAMEO is implemented with LLP.

VI. RESULTS AND ANALYSIS

A. Speedup Comparisons

Figure 13 shows the speedup from using 4GB stacked memory as either a hardware-managed cache, or Two Level Memory (Static and Dynamic), or CAMEO (with Co-Located LLT + LLP). We also compare these designs with an idealistic configuration (*DoubleUse*) that uses the 4GB as a hardware-managed cache, but also increases the size of main memory by an additional 4GB. We show the average speedup (*Gmean*) separately for Capacity-Limited workloads and Latency-Limited workloads. The right-most bar marked *Gmean ALL* is the geometric mean for all workloads. On average, Cache provides an improvement of 50%, TLM-Static provides 33%, TLM-Dynamic provides 50%, CAMEO provides 78%, and DoubleUse provides 82%.

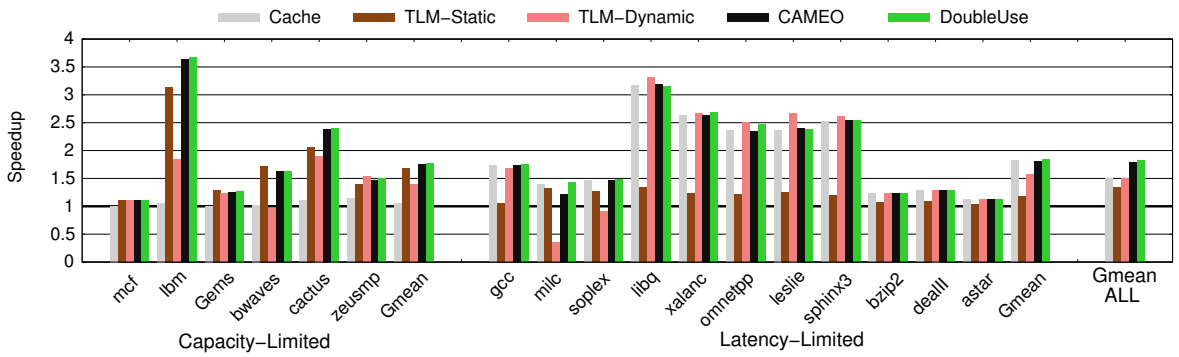

Figure 13. Speedup with stacked memory. CAMEO outperforms both cache and Two Level Memory. CAMEO is close to an idealistic "DoubleUse" design that uses 4GB stacked memory as cache and also increases memory capacity by 4GB.

For Capacity-Limited workloads, CAMEO improves performance by 74%., Cache, on the other hand, improves performance by only 5% because it does not use stacked memory for increasing memory capacity. The performance improvement for Capacity-Limited workloads comes mainly from the increase of memory capacity resulting in fewer page faults. When working set is large, the bandwidth congestion from page migration causes TLM-Dynamic to significantly under-perform TLM-Static.

For Latency-Limited workloads, CAMEO improves performance by 80%, similar to DoubleUse (84%). However, TLM-Static provides only a marginal improvement (18%), as only one-quarter of the requests are serviced by stacked memory. TLM-Dynamic provides 50% on average. For benchmarks that have poor spatial locality (on average, only 10 out of 64 lines in a page get used), such as milc, TLM-Dynamic causes severe slowdown.

In summary, CAMEO provides both memory capacity of TLM, and fine-grained management of cache and hence is able to outperform both designs. On average, the performance of CAMEO is very close to the performance of the idealistic DoubleUse configuration that not only uses the 4GB stacked memory both as hardware cache, and also provides an extra 4GB for memory capacity.

B. Bandwidth Usage in Memory and Storage

Our system contains three modules: stacked DRAM, off-chip DRAM, and storage. An ideal design would reduce the bandwidth consumption of all these three modules simultaneously. However, each of the three designs: Cache, TLM, and CAMEO, optimize the bandwidth of different modules. To measure bandwidth consumption of different designs, we calculate the number of bytes transferred on the bus in respective systems and normalize it to the number in the baseline. Table IV shows the bandwidth usage of stacked memory, off-chip memory, and storage, for different designs, averaged over the workload category.

Cache reduces off-chip bandwidth by 45%, However, Cache does not reduce storage bandwidth. The reason why Cache (and CAMEO) have higher stacked memory bandwidth usage than the baseline is from installs of data lines.

Table IV
BANDWIDTH USAGE IN MEMORY AND STORAGE (CALCULATED AS BYTES TRANSFERRED, AND NORMALIZED TO BASELINE).

	Capacity-Limited			Latency-Limited	
	Stacked	Off-chip	Storage	Stacked	Off-chip
Baseline	n/a	1x	1x	n/a	1x
Cache	1.93x	0.55x	1x	1.76x	0.29x
TLM-Stat	0.26x	0.74x	0.78x	0.25x	0.75x
TLM-Dyn	**2.54x**	**2.19x**	0.78x	1.95x	1.10x
CAMEO	1.89x	1.07x	0.79x	1.51x	0.47x

Both TLM-Static and TLM-Dynamic reduce the bandwidth of storage. TLM-Dynamic consumes significant amount of bandwidth for both off-chip and stacked DRAM due to page migration. Thus, TLM-Dynamic optimizes storage bandwidth at the expense of memory bandwidth.

CAMEO performs a fine granularity transfer between off-chip and DRAM which helps limit the memory bandwidth consumption significantly compared to TLM-Dynamic. The stacked DRAM bandwidth consumption of CAMEO is similar to Cache. However, CAMEO does not provide as much savings as Cache for off-chip bandwidth as it needs to install lines evicted from the stacked to off-chip DRAM. However, unlike Cache, CAMEO does provide a storage bandwidth reduction of 21% for Capacity-Limited workloads.

C. Energy Analysis

We analyze the power consumption and the Energy-Delay Product (EDP) for different designs. The power estimation for DDR3 and storage is derived from [26, 27, 28], and the stacked memory power is estimated based on [29]. For Capacity-Limited workloads, we assume that the processor consumes 60% of the power and the rest is split equally between the storage and memory. For Latency-Limited workloads, we assume processor consumes 70% of the power and memory consumes 30%. Figure 14 shows the normalized power consumption and energy-delay product (EDP) for various designs.

The power consumption increases for all the configurations because of the addition of stacked memory. Overall, Cache increases power by 14%, whereas CAMEO by 37%.

TLM-Dynamic increases power consumption by 51%, as page migration consumes significant power. Cache decreases EDP for Capacity-Limited workloads, because it provides little performance improvement with the addition of stacked memory power. Overall, Cache improves EDP by 4%, and TLM-Static improves by 21%, while CAMEO outperforms all designs by providing 49% EDP improvement.

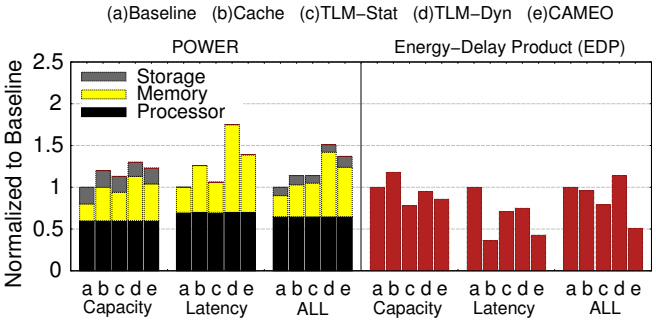

Figure 14. Comparison of Power and Energy-Delay Product. All the numbers are normalized to the baseline system.

D. Optimizing Placement for Stacked DRAM

Both CAMEO and TLM-Dynamic try to migrate recently used data from off-chip memory to stacked memory, albeit at a different granularity. We can improve the overall performance of the system by keeping only the frequently used data in the stacked DRAM. If the OS has oracular knowledge about page access frequencies, it can place the frequently used pages in stacked memory, and thus avoid the overheads of dynamic page migration. We call this idealistic scheme as *TLM-Oracle*. Another approach is to track frequency information on page granularity using dedicated hardware and have the OS periodically perform page migration [18] (termed *TLM-Freq*). Note that TLM-Freq requires significant support from both hardware and OS, as memory access frequency is usually not available to the OS at page granularity.

Figure 15 compares the speedup of CAMEO with different TLM designs. For TLM-Freq, we ignore overheads due to TLB shootdowns, and the software overheads of sorting pages based on access frequencies and performing migration (the bandwidth for page transfer is modeled). For Capacity-Limited workloads, performing migration at page granularity hurts performance. However, for Latency-Limited workloads, with small capacity (<4 GB), the page-based scheme ensures all the frequently accessed pages get accommodated in stacked DRAM. The conflict misses for CAMEO results in performance gap compared to TLM-Freq.

On average, CAMEO provides 78% performance and TLM-Freq provides 61%. Thus, CAMEO outperforms TLM-Freq without the need for page access frequency information, and software support for sorting and page migration. Nonetheless, the two optimizations are orthogonal and can

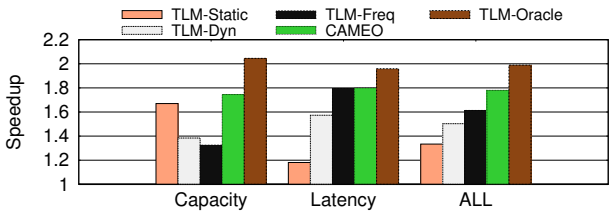

Figure 15. Speedup from optimized page placement in TLM. CAMEO outperforms frequency-based page placement without requiring the tracking support.

be combined for further improvement. For example, if page frequency information is available, CAMEO can retain lines from only heavily used pages in stacked DRAM.

VII. Other Related Work

We use the recently proposed Alloy Cache [11] to implement our caching structure. Several other studies [10, 12, 13, 9, 8] have looked at optimizing DRAM caches. Unfortunately, hardware caches give up on the ability of stacked DRAM to contribute towards OS visible memory-address space.

Optimizing page placement in heterogeneous memories has also been an active area of research [17, 14, 15, 18]. However, as data migration decisions are done at page granularity, such schemes are inefficient at using memory bandwidth. Enhanced page placement policies based on frequency information incur significant implementation complexity from both hardware and software. Our proposal obviates these overheads and still outperforms such schemes.

A recent work from Chatterjee et. al. [30] proposed a heterogeneous memory system that combines RLDRAM and LPDRAM. The first word of a line is stored in RLDRAM and the remaining in LPDRAM. The total capacity of the memory system in their proposal is still equal to the size of LPDRAM, with RLDRAM used only for latency improvement. Furthermore, their design requires significant changes to the on-chip caches (sub-sector valid bits for L1 and L2) and ability of memory controller to send two time-delayed responses back to the core. CAMEO does not require changes to the on-chip cache or to the response path from the memory controller to the core.

Cache Only Memory Architecture (COMA) [31, 32] and Cache-Coherent Non-Uniform Memory Architecture (CC-NUMA) [33, 34] were proposed to do efficient data migration in a shared-memory computer systems consisting of multiple nodes employing non-uniform memory access. It is unclear what such optimizations mean in our context of a single node system with uniform memory access (albeit with two levels of main memory). RAMPage [35] proposed to use SRAM as part of the memory and expose the SRAM capacity to software. However, RAMPage is similar to TLM-Dynamic in that it transfers data at page granularity and requires software support for handling data migration.

VIII. Summary

When the size of stacked DRAM is a significant fraction of the total memory capacity, we want the stacked DRAM to account for OS-visible memory address space, and still retain the fine-granularity and OS-transparent data migration of caches. This paper makes the following contributions:

1) We propose a *Cache-like Memory Organization (CAMEO)* that obtains the best of both worlds: main memory and cache. CAMEO exposes the capacity of stacked DRAM to the OS so that it can count towards memory address space. CAMEO also performs line-granularity data migration transparently, in a manner similar to hardware caches.
2) CAMEO relies on swapping of recently used data lines from off-chip memory to stacked memory. We propose a simple and practical *Line Location Table (LLT)* for CAMEO to track the physical location of all data lines.
3) For data lines resident in the off-chip memory, the performance of CAMEO can be improved by removing the serialization latency due to LLT lookup. We propose a low-latency (single-cycle), low-storage overhead (512 bytes), highly accurate (90%) hardware based *Line Location Predictor (LLP)* to predict the physical location of a line.

Our evaluations show that CAMEO provides an average performance improvement of 78%, outperforming alternative design points of hardware cache (50% improvement) and OS-managed two-level memory (33% improvement). The performance of CAMEO is very close to an idealized system that uses 4GB stacked memory both as a hardware cache and also increases the memory capacity by an additional 4GB.

Acknowledgements

We thank Nagi Aboulenein, Pete Vogt, and Prashant Nair for comments and feedback. This work was supported in part by NSF grant 1319587 and the Center for Future Architecture Research (C-FAR), one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

References

[1] *1Gb-DDR-Mobile-SDRAM-t68: Mobile Low-Power DDR SDRAM: Rev. G 9/11 EN*, Micron, 2010.
[2] *HMC Specification 1.0*, 2013. [Online]. Available: http://www.hybridmemorycube.org
[3] S. Raoux *et al.*, "Phase-change random access memory: a scalable technology," *IBM J. Res. Dev.*, 2008.
[4] M. K. Qureshi *et al.*, "Phase change memory: From devices to systems," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 4, 2011.
[5] JEDEC, *High Bandwidth Memory (HBM) DRAM (JESD235)*, JEDEC, 2013.
[6] *1Gb_DDR3_SDRAM.pdf - Rev. I 02/10 EN*, Micron, 2010.
[7] Micron, *HMC Gen2*, Micron, 2013.
[8] D. Jevdjic *et al.*, "Die-stacked dram caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache," in *ISCA-41*, 2013.
[9] X. Jiang *et al.*, "CHOP: Adaptive filter-based dram caching for CMP server platforms," in *HPCA-16*, 2010.
[10] G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked dram caches," in *Micro-45*, 2011.
[11] M. K. Qureshi *et al.*, "Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design," in *MICRO-45*, 2012.
[12] J. Sim *et al.*, "A mostly-clean dram cache for effective hit speculation and self-balancing dispatch," in *MICRO-45*, 2012.
[13] L. Zhao *et al.*, "Exploring DRAM cache architectures for CMP server platforms," in *ICCD-25*, 2007.
[14] F. Bellosa, "When physical is not real enough," in *the 11th workshop on ACM SIGOPS European workshop*, 2004.
[15] X. Dong *et al.*, "Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support," in *Supercomputing*, 2010.
[16] 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.
[17] H. Huang *et al.*, "Design and implementation of power-aware virtual memory," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2003.
[18] G. H. Loh *et al.*, "Challenges in heterogeneous die-stacked and off-chip memory systems," in *3rd Workshop on SoCs, Heterogeneous Architectures and Workloads*, 2012.
[19] *AMD Phenom II*. [Online]. Available: http://www.amd.com/us/products/phenom-ii
[20] N. P. Jouppi and S. J. Wilton, "Tradeoffs in two-level on-chip caching," in *ISCA-21*, 1994.
[21] *DDR4 SPEC (JESD79-4)*, JEDEC, 2013.
[22] C.-K. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.
[23] A. Hartstein *et al.*, "Cache miss behavior: is it sqrt(2)?" in *Computing frontiers*, 2006.
[24] S. M. Khan *et al.*, "Using dead blocks as a virtual victim cache," in *PACT-19*, 2010.
[25] C.-J. Wu *et al.*, "Ship: signature-based hit predictor for high performance caching," in *Micro-44*, 2011.
[26] *TN-46-03 Calculating DDR Memory System Power*.
[27] E. Seo *et al.*, "Empirical analysis on energy efficiency of flash-based ssds," in *Conference on Power aware computing and systems*, 2008.
[28] Sandisk solid state drive. [Online]. Available: http://www.sandisk.com/products/ssd/
[29] J. Bolaria, "Micron reinvents dram memory," *Microprocessor Report*, 2011.
[30] N. Chatterjee *et al.*, "Leveraging heterogeneity in dram main memories to accelerate critical word access," in *MICRO-45*, 2012.
[31] F. Dahlgren and J. Torrellas, "Cache-only memory architectures," *Computer*, vol. 32, no. 6, pp. 72–79, Jun. 1999.
[32] L. Noordergraaf and R. van der Pas, "Performance experiences on sun's wildfire prototype," in *Supercomputing'99*. New York, NY, USA: ACM, 1999.
[33] B. Falsafi and D. A. Wood, "Reactive numa: A design for unifying s-coma and cc-numa," in *ISCA-24*. New York, NY, USA: ACM, 1997, pp. 229–240.
[34] J. P. Singh *et al.*, "An empirical comparison of the kendall square research ksr-1 and stanford dash multiprocessors," in *Supercomputing'93*, 1993.
[35] P. Machanick *et al.*, "Hardware-software trade-offs in a direct rambus implementation of the rampage memory hierarchy," in *ASPLOS-8*, 1998.