Toward Efficient Programmer-managed Two-level Memory Hierarchies in Exascale Computers

Mitesh R. Meswani Gabriel H. Loh Sergey Blagodurov
David Roberts John Slice Mike Ignatowski

AMD Research
Advanced Micro Devices, Inc.
{mitesh.meswani, gabriel.loh, sergey.blagodurov, david.roberts, john.slice, mike.ignatowski}@amd.com

***Abstract*—Future exascale systems will require very aggressive memory systems simultaneously delivering huge storage capacities and multi-TB/s bandwidths. To achieve the bandwidth targets, in-package, die-stacked memory technologies will likely be necessary. However, these integrated memories do not provide enough capacity to achieve the overall per-node memory size requirements. As a result, conventional off-package memory (e.g., DIMMs) will still be needed. This creates a "two-level memory" (TLM) organization where a portion of the machine's memory space provides high bandwidth, and the remainder provides capacity at a lower level of performance. Effective use of such a heterogeneous memory organization may require the co-design of the software applications along with the advancements in memory architecture. In this paper, we explore the efficacy of programmer-driven approaches to managing a TLM system, using three Exascale proxy applications as case studies.**

I. Introduction

The increasing demand for computational power in scientific, high-performance applications is pushing computer systems into the Exascale regime. Beyond raw floating point performance (e.g., ExaFLOP computing), the government agencies have outlined other significant challenges on the way to practical Exascale machines [28]. In particular, a highly-aggressive memory system is needed to feed the central compute engines. Table I lists the projected memory requirements. At ∼100,000 nodes for the entire system, the requirements imply (at least) 1TB of main memory per node.

To hit bandwidth target, we assume that some form of die-stacked/in-package memory technology is needed [5, 11, 14, 18]. However, our projections indicate that die-stacking alone is unlikely to provide sufficient capacity to satisfy the 1TB capacity requirement. As such, we envision system architectures that employ two (or more) levels of memory. The first-level in-package memory provides the bandwidth, and the second-level off-package memory provides the capacity.

Given a *Two-Level Memory* (TLM) organization, the challenge is then how to best manage such a system. The goal is for the overall system to simultaneously provide the capacity of the slower second-level memory while uniformly providing the high-bandwidth of the faster first-level memory, i.e., the illusion of a single *large and fast* memory. To achieve this, however, some combination of hardware, software, or both, will more than likely be needed to bridge the disparities between the two types of memory.

Ideally, the application programmer should not have to do anything to reap the benefits of a large-and-fast memory. However, user-transparent approaches may require substantial hardware or operating system (OS) modifications. Furthermore, such programmer-transparent mechanisms (whether hardware or software) cannot easily benefit from programmer-level knowledge of the program's data access patterns, reuse characteristics, and other information that may be critical in effectively managing the TLM. In this work, we consider two programmer-transparent TLM organizations (one hardware, one OS), and we also explore the potential for programmer-driven co-design as a way to manage TLM-based systems.

Exascale System	Goal
Performance	1000 PF LINPACK
Power Consumption	20 MW
Total Memory	128 PB
Node Memory Bandwidth	4 TB/s

TABLE I: Exascale system goals as described in the U.S. Department of Energy Exascale System Challenges RFI [28].

II. Exascale Memory Requirements

At an estimated system size of about one-hundred thousand nodes [29] and an overall requirement of 128PB [28], each node will need ∼1TB of memory. At the same time, the targeted bandwidth is an extremely aggressive 4 TB/s. Below, we review our technology assumptions and projections to justify an exascale memory system organization consisting of (at least) two levels of memory.

A. Achieving 4 TB/s of Bandwidth

Current memory technologies, e.g., DDR3, provide only a few tens of GB/s of bandwidth. To achieve the 4 TB/s bandwidth target, we would need hundreds of independent memory channels, whereas typical systems today only have two to four channels. Supporting hundreds of channels would be infeasible due to package pin-count limitations and the corresponding power consumption.

Die stacking is an emerging technology that enables memory chips to directly integrated into the same package as the processor. Figure 1 shows two different stacking technologies with (a) memory stacked on a processor, and (b) memory and processor stacked side-by-side on a silicon interposer. Die-stacking allows the processor to be directly connected to the memory without conventional I/O circuitry or package pin-count limitations, and with shorter-distance (lower impedance) in-package interconnects. As a result, substantially higher bandwidths may be achieved [4, 11, 14, 18, 23].

As an example, the recently published JEDEC "High Bandwidth Memory" standard provides 256 GB/s of bandwidth from a single memory stack [14]. The stack has eight independent 128-bit channels with a 1Gbps transfer rate per pin. By placing eight stacks alongside the processor (Figure 2), one achieves a total bandwidth of 8×256GB/s = 1 TB/s of

978-1-4799-7564-8/14 $31.00 © 2014 IEEE
DOI 10.1109/Co-HPC.2014.8

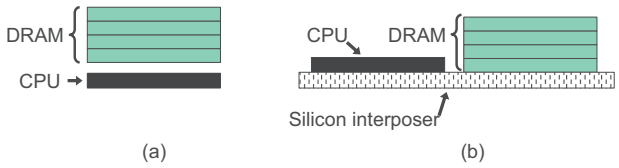

Figure 1: Stacked memory arrangements: (a) vertical 3D stacking, (b) interposer-based 2.5D stacking.

Design Option	Capacity of Eight Stacks (GB)
Current HMC	16
⇒ 8Gb DRAM dies	32
⇒ Eight layers per stack	64
⇒ 16Gb DRAM dies (??)	128
⇒ Sixteen layers per stack (??)	256

TABLE II: Capacity scaling of in-package memory.

bandwidth. We believe it is reasonable that by around the end of the decade, an additional factor of four (or more) in HBM bandwidth is achievable through a combination of wider buses, more channels, and/or faster data rates.[1] The main conclusion is that, to even have a chance at achieving the exascale memory bandwidth targets, *die-stacked memory must be an integral part of an exascale memory solution*.

B. Achieving 1TB of Per-Node Capacity

Current die-stacked memories provide about only 1-2 GB of memory per stack. For example, Micron has released Hybrid Memory Cube (HMC) samples consisting of four layers of 4Gb DRAM dies, providing a total capacity of 2GB [22]. Starting from here, where eight stacks provide a total of 16GB, Table II shows a possible path for scaling a node's memory capacity.

The first two scaling options of using denser 8Gb layers and a total of eight layers of DRAM are reasonably low-risk approaches: the existing DDR3 memory standard already defines up to an 8Gb DRAM device, and various die-stacked DRAM prototypes have been demonstrated with eight layers of memory [15]. The following two scaling points (indicated with "??") represent more speculative options with greater risk of not being available by exascale time frames. Continued scaling of DRAM to 16Gb devices requires overcoming an array of manufacturing and design issues [10]. Building memory stacks of increasingly taller heights runs into power delivery, thermal, yield, mechanical assembly, testability, and other challenges [9, 17].

Even if we optimistically assume that the various technology scaling problems are taken care of, the capacity projection shown in Table II only gets us to one fourth of the target 1TB capacity per node. Based on this analysis, we conclude that *die-stacking alone is insufficient for an exascale memory solution*.

C. The Case for a Two-Level Memory

From the above analysis and projections, we conclude that (1) die-stacked memory is necessary, and (2) die-stacked memory is insufficient. This leads to a memory system design that includes as much die-stacked memory as possible to provide a first level of memory that meets the exascale bandwidth requirement, coupled with a second level of off-package memory to supply the necessary per-node capacity. Figure 2 illustrates such a Two-Level Memory (TLM) system for a single node in an exascale system. Note that the viability of such a memory organization is predicated on the assumption that through some means, whether hardware and/or software, we can make such a heterogeneous memory perform as if it were a single, large, and flat layer of uniformly fast, high-bandwidth memory. In the remainder of this paper, we explore different approaches to managing a TLM system, the role of co-design in the solution, and discuss directions where further research and co-design efforts are required.

[1] Adding more than eight separate stacks to further improve bandwidth is challenging as the overall silicon interposer size required for such a system would rapidly exceed the reticle limit.

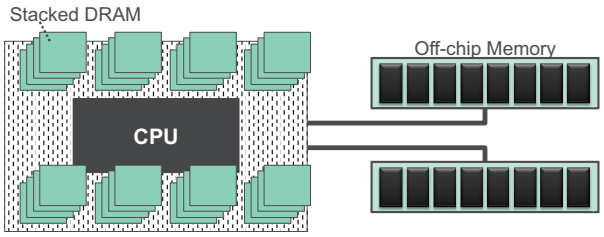

Figure 2: Organization of a processor with two levels of memory. The first level consists of die-stacked DRAM; the second level is off package and consists of multiple packages of memory, possibly non-volatile memory.

III. TLM Usage Models

A. Programmer-transparent Models

For comparison, we consider two programmer-transparent usage models for a TLM system, one hardware based and the other software based.

Hardware Cache: There has been a significant amount of recent research on hardware-based techniques to make use of die-stacked DRAM as a very large, last-level cache [6, 8, 12, 13, 19, 21, 24, 26, 30]. The advantage of such approaches is that the software (both application and operating system) need not be modified to receive the benefits of this additional cache. The difficulty arises from implementation challenges primarily around building tag arrays capable of tracking multiple gigabytes worth of stacked DRAM data, and the efficacy of insertion and replacement policies.

The tag array problem is primarily one of implementation cost and complexity. With the recent research results from the computer architecture community, it is reasonable to assume that viable solutions exist. The second problem regarding the management of the cache (i.e., what should be cached?) is more difficult and hints at an area where co-design may be of value. Studies have shown that the locality of the reference stream at the level of a stacked DRAM cache (i.e., L3 or even L4) is quite different from what is observed in an upper-level cache (e.g., L1). In particular, any temporal locality of "hot" cache lines would already be absorbed by the L1 and L2 caches, and therefore accesses at the L3 and L4 levels are "cool" or "luke-warm" at best. This makes it much more difficult for the hardware cache controllers to decide which lines should be evicted or which lines should even be cached in the first place. An application programmer could potentially have much more information about the access patterns of particular data objects and make better decisions about what items should be placed in the fast memory versus off-package memory.

OS-managed Page Cache: Another approach is to have the operating system manage the TLM, but continue to hide the memory heterogeneity from the user-level/application. One means of implementing such a system is to make use of the existing virtual memory paging mechanisms. The fast, in-package memory can be used like a conventional "main memory" where loads and stores can directly access it with the aid of page-table/TLB mappings from virtual addresses to physical addresses in the stacked DRAM. The slower, off-package memory acts like a conventional swap device (except *many* orders of magnitude faster than disk). Memory accesses to virtual addresses not currently mapped to the stacked DRAM cause a fault that the operating system handles, where it copies the requested page from the off-package memory to the fast stacked DRAM, updates the page tables to reflect the new mappings, and then allows the application to continue execution. Because even off-chip memory is so much faster compared to conventional disk-based swap, the overhead of this kind of TLM swap operation will be dominated by the interrupt handling and TLB shoot down latencies rather than the raw data transfer between memory levels.

The advantage of an OS-driven TLM approach is that the hardware remains very simple (practically no changes are needed; the stacked DRAM is simply exposed to the OS as another region of the physical address space in a somewhat NUMA-like fashion). The disadvantages are that the interrupt and TLB shoot down latency for handling a page fault (first-level memory miss) is orders of magnitude slower than the hardware cache-fill mechanism, and, similar to the hardware cache, the OS has very limited knowledge about the application's memory usage patterns which could be useful in selecting replacement candidates from the stacked DRAM or prefetching additional pages to amortize the cost of the page-fault handler.

B. User-directed TLM Management

In contrast to the above TLM usage models, the other approach is to simply expose the existence of the multiple memories to the application programmer, and let him/her deal with it. Similar to the OS's NUMA-like view of the physical memory described above for the page cache, the different memory regions would also be made explicit to the programmer. There are many ways that this can be exposed to the programmer (and we discuss a simple `malloc`-based approach below), but ultimately the programmer is responsible for placing/allocating data objects across the different types of memory.

The hardware implementation advantages are similar to the OS page cache in that very little support is required. Memory accesses to different address ranges simply map to the different levels of memory. The OS must provide some interfaces and support to allow the programmer to specify desired levels of memory for allocations. The standard virtual memory interface (i.e., page tables) can track which virtual addresses map to fast, die-stacked memory or to slower off-package memory; an example schematic representation of such a page table-based mapping is shown in Figure 3.

The application programmer now not only has the ability to dictate whether a data object should be placed in one level of memory or the other, but he/she also has the responsibility to do so. This approach enables fine-tuning of memory system performance through the orchestration of the memory placement and migration, but the cost is that the required level of programmer involvement is much higher.

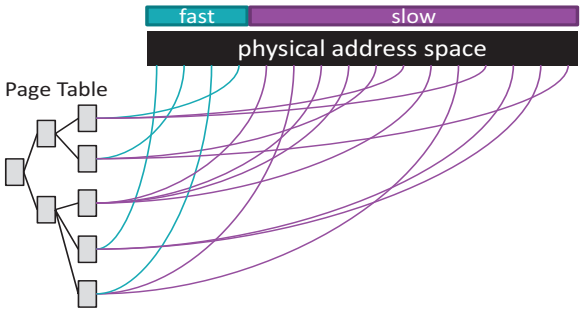

Figure 3: Depiction of mapping of a process's virtual address space to a physical address space that spans both levels of memory.

Previous work suggested that having some mechanism other than a purely reactive, hardware-caching approach has greater overall performance potential for TLM systems [20]. The prior study considered a first-level memory capacity of N pages with a simple static approach that chose the top-N pages responsible for the most main memory traffic and pinned these in the first-level memory. The study showed that this simple approach provided superior performance compared to a hardware-based cache. While that study assumed some form of oracular or off line analysis to select the N hottest pages (and therefore does not provide a practical solution), it strongly suggests that the performance *potential* of higher-level (i.e., something smarter than a simple cache) TLM management schemes is worth further investigation. It is intuitive that a programmer's deep knowledge of an application's behavior should be helpful in co-design an effective TLM solution.

TLM Malloc: The TLM malloc API is provided as an example interface for the application writer to provide directives about the placement of data structures or memory objects in the TLM system. In this work, we make use of a simple `malloc` interface to allow the programmer to explicitly specify whether memory allocations are placed in the first-level (die-stacked) or second-level (off-chip) memories. In our studies, we make use of the following three function calls:

```
void * mallocLevel1(size_t num_bytes)
      /* Allocate num_bytes from 1st-level memory */
void * mallocLevel2(size_t num_bytes)
      /* Allocate num_bytes from 2nd-level memory */
bool reloc(void * obj, int new_level)
            /* Relocate obj to new memory level */
```

In our experiments, the allocation is successful only if there is enough capacity for the entire object at the desired memory level or else it fails (i.e., returns NULL). There are other alternatives possible such as using allocation as hints (e.g., allocate in fast memory if possible, but fall back to slow memory if necessary). The industry has already begun thinking of implications of mallocing data to a second-level persistent storage [25] where the mallocs are persistent across different program run times and possible machine reboots. In these cases, malloc may want to attach to regions that were persistent across previous runs. The API that we use in our study is not meant to be a complete, mature solution, but rather to serve as a working model to begin exploring the issues around programmer-managed TLM systems.

IV. Reigning in User-driven TLM Management

The programmer effort required to manually manage a user-exposed TLM system (i.e., NUMA-like) may vary from minimal to intractable. Some applications have a few key data structures with easily understood access patterns that lend themselves to straightforward partitioning between the levels of memory. In other applications, there may be many thousands of individual allocations, making an optimal assignment of objects to memory levels combinatorially difficult. In this section, we discuss policies and mechanisms to reduce the burden on the programmer when manually optimizing an application for a TLM system.

A. Selective Pinning

Using our TLM-malloc interface (Section III-B), the application programmer can explicitly allocate or *pin* specific data objects into one level of memory or the other. In this work, we make use of a profile-guided approach to assist the programmer in making these decisions; however any method or heuristic could be employed. However, a conventional malloc call can still be used, which indicates that the application programmer has no specific preference for how the object should be placed within the TLM hierarchy. The reasons for this indifference could be due to low importance (e.g., the object is not frequently accessed and so placement of this object is not critical to performance), performance "ignorance" (i.e., the programmer is unsure of the performance sensitivity of the object and chooses to not make a decision one way or the other), or simply as a matter of programming effort (i.e., this object could potentially be important, but the programmer did not have the time to analyze whether it should be pinned to fast memory).

Below, we discuss two approaches for using application profiling to aid the programmer in selecting objects to pin to the fast memory. Later, we will discuss OS support to handle the remaining memory objects not explicitly placed by the programmer.

Access-rate Profiling: We developed a tool (see Section V) to profile an application so that every memory access can be attributed back to the original data structure in the source code. In particular, we only track memory requests that generate traffic to *main memory*; i.e., accesses that hit in the on-chip caches do not contribute toward the profiled traffic. As a result, we can generate a profile that lists, for every object in the program (at a filename-line number level of granularity), exactly how much read and write traffic in bytes that the object was responsible for. By simply analyzing the resulting profile, the programmer can select the top-N objects to pin into the fast, die-stacked DRAM (where N is simply determined by the number of objects that one can place into fast memory before running out of capacity).

Access-density Profiling: One potential shortcoming of ranking objects based on absolute memory traffic is that it does not directly capture the amount of reuse or locality. For example, a 1GB array that is sequentially accessed only once may generate a total of 1GB of traffic to main memory, but pinning this into the fast memory may provide relatively little performance benefit. As such, we also make use of a second profiling heuristic where we profile and rank source-level objects based on the total memory traffic for the object, divided by the size of the object's allocation. This provides the programmer a list of the objects with the greatest levels of reuse, and therefore (hopefully) provides more efficient usage of the limited fast memory capacity.

Limitations: The profiling techniques used in our studies require non-trivial overheads. We manually instrument the target applications and run them in a system emulator that includes a cache-hierarchy simulation to ensure that we profile only the post-cache traffic that would impact the usage of the TLM portion of the memory hierarchy. Further research is required to develop lighter-weight tools and methodologies to capture useful program profiles for TLM optimization, but such efforts are left for future work.

B. What About the Rest of Memory?

Using selective pinning, the application programmer only needs to invoke the TLM-malloc API for the subset of data structures and memory objects that he/she deems to be important. For the remaining memory objects (including the function call stack, the code segments, and other automatic memory items), we assume a simple OS-based mechanism. We describe a simple scheme, although more research is needed to develop more effective algorithms.

One challenge for the OS in managing a NUMA-like TLM organization is that it has limited visibility into the memory access patterns of the application. In particular, after an initial page fault, the OS does not "see" any further memory traffic to a page so long as the page remains mapped and valid (i.e., CPU loads and stores hit in the TLB or page table walk and access physical memory directly without any OS intervention or interaction). To deal with this, we make use of a simple epoch-based "first touch" (FT) heuristic. We divide program execution into separate epochs (e.g., 100 million cycles). At the start of each epoch, we mark all non-pinned pages as invalid in the page table. Any subsequent access to a non-pinned page generates a page fault (due to an invalid page table entry or PTE). If the page is already in the fast die-stacked memory, the OS sets the page's PTE to valid, and then execution continues. If the page is currently in the off-package memory, the OS migrates/copies the page to fast memory (evicting an invalid page if necessary), updates its PTE to reflect the new location, and then execution resumes. After a certain number of page validations or migrations, the fast memory will eventually run out of capacity for any additional pages. At this point, all remaining pages stay put in the slower off-package memory, the OS marks all of their PTEs as valid (this can be done incrementally or lazily), and then execution resumes for the remainder of the epoch.

This is a very simple approach, but it allows useful memory to have a chance at being brought into the fast die-stacked DRAM, and unused/cold pages to migrate back out to the off-package memory. More sophisticated algorithms are possible, especially if JIT recompilation techniques are employed, if the hardware provides additional monitoring/profiling support, etc. We expect this to be a fruitful area for new research.

V. Experimental Methodology

A. System Emulation with SimNow

To evaluate the effectiveness of different TLM management approaches, we implemented a TLM simulator (TLMsim) on top of AMD's SimNow™ tool. SimNow™ is an AMD64 technology-compatible x86 platform emulator for AMD's family of processors [3]. It functionally models a complete computer system including the processing cores, devices, the OS,

Name	Description
LULESH	Shock Hydrodynamics Challenge Problem
CoMD	Molecular dynamics proxy application suite
XSBench	Represents a key computational kernel of the Monte Carlo neutronics application OpenMC.

TABLE III: The workloads evaluated in this study.

Benchmark	Sizes	TLM Ratio L1:L2	MPKI Ratio Simulated vs. Real
Lulesh	-s 150 -i 10	256MB:2GB	3.28
CoMD	-x 55 -y 55 -z 55 –nSteps 20	64MB:512MB	0.78
XSBench	-s small -g 90750	256MB:2GB	0.38

TABLE IV: Workload configurations.

and the application program of interest. We modified SimNow to include a simulation of a cache hierarchy, and we instrument SimNow to capture all traffic to main memory as it misses in the last-level cache. From here, we feed the main memory traffic into a TLM system simulator that models the two different types of memory, along with the different management policies (e.g., hardware cache, programmer-directed pinning, etc.). We added an API that allows the emulated target application to call the TLM-malloc functions, which SimNow can intercept and provide the allocation information to the back-end TLM simulator. All reported results were collected from runs that simulated a 32-core system.

B. Benchmarks

In our evaluations, we make use of three HPC proxy applications [1]. In particular, we use the apps listed in Table III, which include LULESH [16], XSBench [2], and CoMD [27]. We selected these three as they have relatively higher levels of memory traffic. While our SimNow-based infrastructure provides relatively fast system-level simulation, it is still slower than running applications on native hardware by several orders of magnitude. As a result, our evaluations use smaller problem sizes (Table IV) to maintain tractable simulation run times.

To minimize the impacts of reduced problem sizes on the overall representativeness of the results, we calibrated the sizes of each our modeled on-chip caches such that the overall memory access rate (measured in average misses per thousand instructions (MPKI)) from our simulations were of a similar order of magnitude compared to the rates observed from hardware performance counters when executing the applications on native hardware at normal problem sizes. As each application has a different working set size, we maintained the same approximate *ratio* of the size of fast die-stacked DRAM (L1) to the size of the slower off-package memory (L2). The actual modeled sizes are shown in Table IV is about 1:8 (fast:slow).

C. TLM Configurations

We evaluate the three different TLM usage models described in Section III. The configuration details are summarized in Table V. The hardware-cache uses a direct-mapped organization [24] with 64-byte cache lines, and we use hardware prefetching to pro-actively load more than one cache line per miss. We evaluated different prefetch amounts (up to loading 1KB at a time), and in the reported results we use the setting that provided the best hit rate for each workload. The OS-managed page cache brings pages into the fast memory whenever the page is not already present, and selects pages for

Name	Description
Hardware Cache	Direct-Mapped cache with 64 byte line size with prefetching between 128B to 1KB on a miss
Page Cache	First-level memory managed as an OS page cache, 4KB pages
User-directed Pinning	Explicit memory placement via TLM-malloc API, undeclared pages managed with first-touch policy

TABLE V: TLM management approaches evaluated in this work.

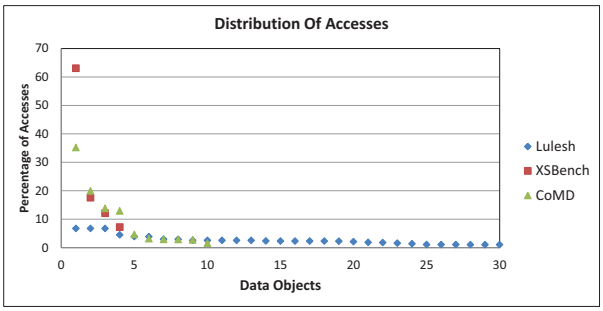

Figure 4: The thirty program objects responsible for the greatest amount of total main memory traffic, and the percentage of main memory traffic attributable to each.

eviction using a multi-queue replacement policy [7]. Finally, we consider the case where the programmer explicitly pins memory objects using the TLM-malloc interface, with any undeclared objects being managed with a first-touch policy as described in Section III-B. We make use of both access-rate and access-density profiling heuristics to select which data structures should be pinned. For comparison, we also consider a case where *no* objects are explicitly pinned, thereby leaving all of memory to be managed using the first-touch heuristic.

D. Performance Metrics

To gain insight into the impact of the different TLM-management approaches, we consider two metrics. The first is the hit rate of the first-level memory, which is simply the fraction of all memory requests (loads and stores) that miss in the cache hierarchy, but are successfully serviced from the fast, die-stacked DRAM. The second metric is the total traffic (including demand misses, migration, and write back operations) to each of the memory levels, which serves as a proxy for the bandwidth overheads of each respective approach.

VI. Application Characterization

We next present results from profiling the memory characteristics of the evaluated proxy applications. To help with our pinning studies, we analyze traffic by access rate and by access density, corresponding to the pinning heuristics described in Section IV.

LULESH Characteristics: In LULESH, memory traffic is spread relatively pretty evenly across more than twenty distinct data objects. Figure 4 shows the top thirty objects for each benchmark, sorted from the one that is responsible for the most post-cache main memory traffic to the 30th-most. Based on this even distribution, we do not observe a key set of "hot" data objects that are obvious candidates for pinning into the fast die-stacked DRAM based on this profiling metric.

Figure 5 shows the top thirty objects in LULESH when ranked by accesses-per-byte. Using this metric, we can identify four data structures that have significantly higher levels of

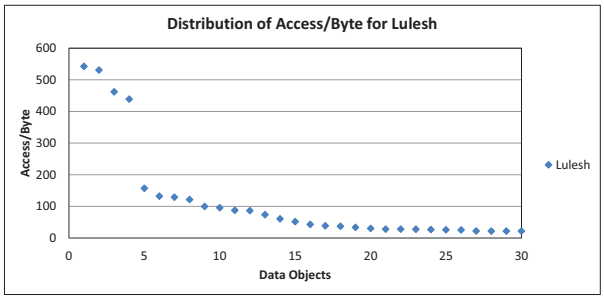

Figure 5: The top thirty program objects in LULESH when ranked by main memory accesses divided object size.

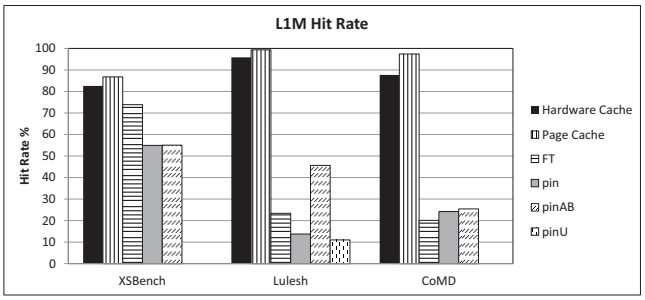

Figure 6: Hit rates in die-stacked, level-one memory. "pin" corresponds to pinning based on access-rate profile, "pinAB" is based on access-density profiles, and "pinU" is based on user feedback, which is only for LULESH.

reuse than the others, which would potentially make for good candidates for pinning.

As it currently stands, our profiling techniques are fairly invasive and heavy-weight. One question of interest is whether or not the application programmers/domain experts could provide the same or similar information without needing invasive instrumentation and profiling. As a case study, we consulted with LULESH experts who provided us a list of objects in three categories based on whether the objects persist across iterations/physics packages. The idea is that objects that are passed from one phase of a simulation to the next would be good candidates for pinning in the first-level memory. Several of their "always used" objects also exhibit high levels of reuse based on our profiling data, which indicates that the programmers were able to identify some of the good pinning candidates. However, several of their always-used objects did not make it on to our top-30 list. Also interesting is that several objects from our profiling are found in their "never reused" list. In addition to our profile-based heuristics, we also evaluate the effectiveness of using the expert-guided pinning recommendations.

XSBench and CoMD Characteristics: In contrast to LULESH, XSBench has three data objects that account for more than 95% of the application's overall memory traffic. From these observations, we can conclude that for pinning purposes, it is sufficient to pin these three data objects into the first-level memory (if they fit). CoMD has 4 objects that account for 80% of accesses, and the top 12 objects are required to cover 95% of accesses. Similar to XSBench, we can cover majority of CoMD's access by pinning the top four objects in the fast die-stacked DRAM.

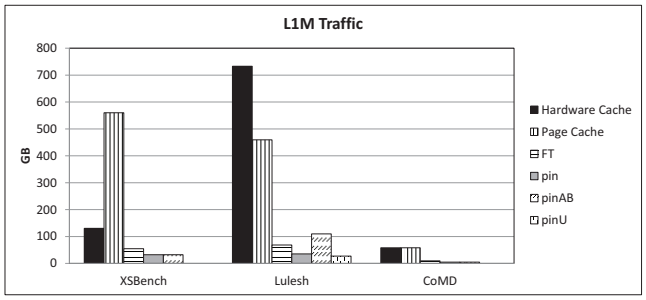

Figure 7: Total traffic at the die-stacked, level-one memory.

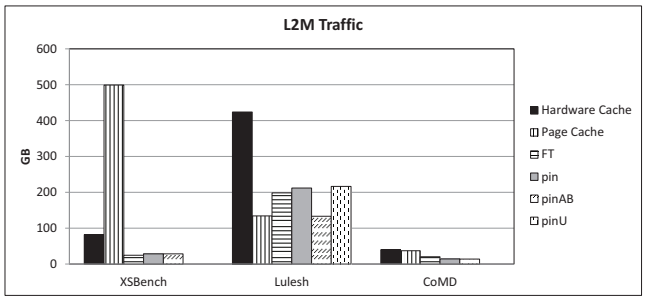

Figure 8: Total traffic at the off-package, level-two memory.

VII. Experimental Results

A. Hardware and OS Caching

We first analyze the hit rate at the level-one memory (die-stacked DRAM) for our three applications according to the different TLM management schemes. For the hardware cache, we only show results corresponding to the best prefetching settings, which were 256 bytes, 1KB, and 512 bytes for XSBench, LULESH, and CoMD, respectively. Figure 6 shows the fast die-stacked DRAM (L1M) hit rate for the three applications. Looking strictly at hit rate, both the hardware cache and the OS-managed page cache provide the best results. The programmer-/profile-based approaches to manually manage the placement of objects across the two levels of memory (pin*) does reasonably well for XSBench, but has significantly lower hit rates for LULESH and CoMD.

At first blush, the hit-rate results might suggest that it is unnecessary to have the programmer spend much effort in manually co-designing his/her application to run on a TLM system. However, these programmer-transparent caching approaches have other overheads associated with them. Figure 7 and Figure 8 show the amount of memory traffic to the first-level (fast-DRAM) and second-level (off-package) memories, respectively. For the total traffic to the fast-DRAM, the OS-managed page cache consumes the most bandwidth in case of XSBench. This is because each time there is a miss in the first-level memory, an entire 4KB page must be brought in from the second-level, off-package memory, even if the application only needs a single byte from the entire page. Similarly, if even a single byte on a page has been modified, the entire page must be written back to the second-level memory as the OS's page tables do not track modifications at a sub-page granularity.

In comparison to the OS-managed page cache, the hardware cache does not generate nearly as much activity to the first-level memory for XSBench (while maintaining competitive hit

rates). A primary reason for this is that our hardware cache approach fetches and stores cachelines on a 64B granularity. While our prefetching may bring in multiple cachelines per miss, the amount is still less than the entire 4KB page. Furthermore, the hardware cache maintains dirty-state on a cacheline granularity, and so only individual modified cachelines generate writeback traffic upon eviction. LULESH is an interesting case where, despite the high overall hit rates, the hardware cache still generates more traffic to both levels of memory compared to the OS page cache. Part of this may be due to the heavily reduced associativity of a direct-mapped cache organization (contrast to the OS page cache which is effectively fully-associative).

The programmer-driven pinning approaches (discussed in more detail below), however, generally maintain lower levels of memory traffic, but pay for this through lower hit rates in the faster first-level memory.

B. Programmer-driven Pinning

The hit rates in Figure 6 for the programmer-driven approaches are consistently lower than the caching approaches. There are several contributing factors to this. First, for some applications, there are a few key data structures, but each one is often quite large. As a result, the programmer can only choose to pin a limited subset of these before exhausting the capacity of the first-level memory. Second, some of the frequently accessed data structures are not always accessed *continuously* throughout a program's execution. However, pinning such an object into the fast memory consumes the capacity regardless of whether the object is currently being used. The caching approaches, in contrast, can evict items that are not currently being used to better utilize the capacity for other more actively accessed objects.

These results do not invalidate programmer-managed TLM approaches in general, but they do suggest that the programmers will need more sophisticated techniques beyond statically assigning data structures among the levels of memory. For example, data structures may need to be partitioned into smaller components (e.g., a single 4GB array may not fit into 1GB of stacked DRAM, but splitting the array into several smaller pieces may allow the programmer to place the subset of more frequently accessed regions into the fast memory). Also, instead of static pinning, the programmer may need to play a more active role in dynamically migrating/scheduling the memory (e.g., manually "prefetching" objects into fast memory before they are needed by the program, and aggressively migrating them back to the slower off-package memory when not needed in the immediate future).

Figure 6 also shows the hit rate impact when using different heuristics to guide the programmer in selecting objects to pin in the fast memory. The bars labeled "pin" correspond to using the absolute traffic per objects as a way of ranking the most "important" objects, whereas "pinAB" uses the traffic normalized by object size (AB=Accesses-per-Byte). For XSBench and CoMD, both profiling metrics result in similar hit rates, which is not surprising given the characterization in Figure 4 that showed that there were only a few objects responsible for the vast majority of traffic; i.e., either metric will key in on these handful of data structures.

LULESH behaves quite differently in that its accesses are spread out over a much larger number of distinct memory objects. As such, relying purely on absolute traffic does not perform as well. Consider the distribution in Figure 4, where two neighboring objects in the ranking may be responsible for very similar amounts of traffic, but one object may be many times larger. Picking one over the other provides the same traffic benefit *for this one object*, however, choosing the larger one will eventually preclude the inclusion of other objects because the die-stacked DRAM will fill up faster. For LULESH, ranking objects based on the access-density of objects results in about a 3× improvement in hit rate compared to absolute traffic. These results highlight the importance of developing effective tools to help programmers identify the data structures that exhibit the highest levels of reuse.

Incorporating Domain Expertise: For the LULESH application, we also evaluated the efficacy of a programmer-managed TLM system using the expert-selected objects described in Section VI. As was previously discussed, the expert-provided list did not completely overlap with our list from the access-density profile, and as a result the hit rates ("pinU" in Figure 6) are much lower. There are several contributing factors at play here. First, the experts consulted were considering the usage of LULESH in the context of a larger, multi-physics application where pinning objects that persist from one phase of simulation to the next (i.e., between different physics packages) may have a different impact on the fast-DRAM hit rates. We did not have access to these workloads, and so in our LULESH-only scenario, the pinning choices are likely to be different. Another is that the programmer typically thinks in terms of loads and stores, but an object that is frequently accessed at this level may not actually generate a large amount of traffic to main memory (i.e., if those accesses usually hit in the on-chip caches). Training programmers and/or providing them with better tools to reason about memory access locality patterns *in the presence of multi-level cache hierarchies* may be needed for the programmers to effectively manage a TLM organization.

C. Discussion

While the results of the programmer-driven management of the TLM may not be as inspiring as one might initially hope, they are still very interesting as this helps to shine light on several avenues for future research. LULESH's pinning results based on accesses-per-byte provides nearly half of the first-level memory hit rate as the dynamic caching schemes, which is actually somewhat inspiring as one would imagine that research into better pinning support (e.g., through better sizing of data structures, dynamic migration and prefetching, etc.) could close a lot of the gap. For LULESH in particular, the high hit rates from the hardware and OS caching come with a significant cost in bandwidth/over-fetch. Furthermore, the OS-based approach also suffers from significant performance overheads to update the page tables and shootdown TLBs, which in current systems can cost tens of thousands of cycles depending on the number of cores. In contrast, the first-touch heuristic for the unmanaged portions of the TLM is invoked relatively infrequently, leading to fewer than 4% the number of page faults compared to the OS page cache. The hardware cache appears to be fairly attractive, but the implementation complexity and cost for the control logic and tags to manage the cache are not insignificant. We have started preliminary performance modeling of these approaches, and thus far there is no clear winner. Some approaches are more attractive for

some workloads, and there are different hardware, software, and programmer-productivity costs.

VIII. Conclusions

In this paper we compared and contrasted two automated schemes against user managed schemes for managing two-level memory (TLM) hierarchies. Due to several pressing constraints, the memory subsystem of the future will likely contain a heterogeneous mix of memory technologies, thus prompting researchers to devise management policies that migrate workload data across levels of memory on demand. An application programmer's domain knowledge could play a critical role to manage TLM efficiently. Our results indicate that while the automated schemes have higher hit rates in the faster memories, they come at a price of higher bandwidth utilization due to misses and evictions. In contrast, our user-managed TLM policies did fare better with lower bandwidth utilization although their hit rates were also lower. These results motivate additional research to better understand application memory access patterns, methodologies and tools to help programmers better organize their applications to use a TLM system, development of better OS support to handle the regions of memory not explicitly managed by the programmer, hardware support for better profiling and detection of hot memory regions/objects, hardware/OS support for reducing the cost of migrations/page-table updates, and more. This work has taken an initial step toward trying to expose some of the tradeoffs and challenges, but there is still much research for the community to undertake.

References

[1] Co-Design — U.S. DOE Office of Science (SC). http://science.energy.gov/ascr/research/scidac/co-design/.

[2] Proxy-Apps for Neutronics. https://cesar.mcs.anl.gov/content/software/neutronics.

[3] AMD Corp. Amd simnow. simulator 4.6.1. http://developer.amd.com/wordpress/media/2012/10/SimNowUsersManual4.6.1.pdf, November 2010.

[4] B. Black. Die Stacking is Happening. In *Proc. of the Intl. Symp. on Microarchitecture*, Davis, CA, December 2013.

[5] B. Black, M. M. Annavaram, E. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCauley, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. P. Shen, and C. Webb. Die-Stacking (3D) Microarchitecture. In *Proc. of the 39th Intl. Symp. on Microarchitecture*, Orlando, FL, December 2006.

[6] M. El-Nacouzi, I. Atta, M. Papadopoulou, J. Zebchuk, N. E. Jerger, and A. Moshovos. A Dual Grain Hit-miss Detector for Large Die-stacked DRAM Caches. In *Proc. of the Conf. on Design, Automation and Test in Europe*, pages 89–92, 2013.

[7] B. V. Essen, H. Hsieh, S. Ames, and M. Gokhale. DI-MMAP: A High Performance Memory-Map Runtime for Data-Intensive Applications. In *Proc. of the ACM/IEEE Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 731–735, Salt Lake City, UT, November 2012.

[8] F. Hameed, L. Bauer, and J. Henkel. Simultaneously Optimizing DRAM Cache Hit Latency and Miss Rate via Novel Set Mapping Policies. In *Proc. of the*, 2013.

[9] M. Healy and S. K. Lim. A Study of Stacking Limit and Scaling in 3D ICs: An Interconnect Perspective. In *IEEE Electronic Components and Technology Conference*, pages 1213–1220, 2009.

[10] International Technology Roadmap for Semiconductors. Process, integration, devices and structures, 2011. http://www.itrs.net/Links/2011ITRS/2011Chapters/2011PIDS.pdf.

[11] JEDEC. Wide I/O Single Data Rate (Wide I/O SDR). http://www.jedec.org/standards-documents/docs/jesd229.

[12] D. Jevdjic, S. Volos, and B. Falsafi. Die-stacked dram caches for servers. In *Proc. of the Intl. Symp. on Computer Architecture*, 2013.

[13] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian. CHOP: Adaptive Filter-Based DRAM Caching for CMP Server Platforms. In *Proc. of the 16th Intl. Symp. on High Performance Computer Architecture*, pages 1–12, January 2010.

[14] Joint Electron Devices Engineering Council. JEDEC: 3D-ICs. http://www.jedec.org/category/technology-focus-area/3d-ics-0.

[15] J.-S. Kim, C. Oh, H. Lee, D. Lee, H.-R. Hwang, S. Hwang, B. Na, J. Moon, J.-G. Kim, H. Park, J.-W. Ryu, K. Park, S.-K. Kang, S.-Y. Kim, H. Kim, J.-M. Bang, H. Cho, M. Jang, C. Han, J.-B. Lee, K. Kyung, J.-S. Choi, and Y.-H. Jun. A 1.2V 12.8GB/s 2Gb Mobile Wide-I/O DRAM with 4x128 I/Os Using TSV-Based Stacking. In *Proc. of the Intl. Solid-State Circuits Conference*, San Francisco, CA, February 2011.

[16] Lawrence Livermore National Laboratory. Hydrodynamics Challenge Problem. Technical Report LLNL-TR-490254.

[17] D. L. Lewis, S. Panth, X. Zhao, S. K. Lim, and H.-H. S. Lee. Designing 3d test wrappers for pre-bond and post-bond test of 3d embedded cores. In *Proc. of the 29th Intl. Conf. on Computer Design*, pages 90–95, October 2011.

[18] G. H. Loh. 3D-Stacked Memory Architectures for Multi-Core Processors. In *Proc. of the 35th Intl. Symp. on Computer Architecture*, Beijing, China, June 2008.

[19] G. H. Loh and M. D. Hill. Supporting Very Large Caches with Conventional Block Sizes. In *Proc. of the 44th Intl. Symp. on Microarchitecture*, Porto Alegre, Brazil, December 2011.

[20] G. H. Loh, N. Jayasena, K. McGrath, M. O'Connor, S. Reinhardt, and J. Chung. Challenges in Heterogeneous Die-Stacked and Off-Chip Memory Systems. In *3rd Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW)*, New Orleans, LA, February 2012.

[21] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan. Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management. *Computer Architecture Letters*, 11(2):61–64, July 2012.

[22] Micron Technology, Inc. Micron Technology SHips First Samples of Hybrid Memory Cube. Press Release, September 2013. http://investors.micron.com/releasedetail.cfm?ReleaseID=793156.

[23] J. T. Pawlowski. Hybrid Memory Cube: Breakthrough DRAM Performance with a Fundamentally Re-Architected DRAM Subsystem. In *Proc. of the 23rd Hot Chips*, Stanford, CA, August 2011.

[24] M. Qureshi and G. H. Loh. Fundamental Latency Trade-offs in Architecturing DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design. In *Proc. of the 45th Intl. Symp. on Microarchitecture*, Vancouver, Canada, December 2012.

[25] A. Rudoff. The impact of the nvram programming model. http://www.snia.org/sites/default/files2/SDC2013/presentations/GeneralSession/AndyRudoff_Impact_NVM.pdf, 2013.

[26] J. Sim, G. H. Loh, H. Kim, M. O'Connor, and M. Thottethodi. A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch. In *Proc. of the 45th Intl. Symp. on Microarchitecture*, Vancouver, Canada, December 2012.

[27] S. Swaminarayan, J. Mohd-Yusof, and C. Sewell. CoMD: A Molecular Dynamics Proxy Applications Suite. LA-CC 11-119, Los Alamos National Laboratory, November 2012.

[28] U.S. Department of Energy. Request for Information (RFI): Exascale Research and Development. No. 1-KD73-I-31583-00, July 12 2011.

[29] U.S. Department of Energy. Scientific Discovery at the Exascale. Technical report, DOE ASCR 2011 Workshop on Exascale Data Management, Analysis, and Visualization, February 2011.

[30] L. Zhao, R. Iyer, R. Illikkal, and D. Newell. Exploring DRAM cache architectures for CMP server platforms. In *Proc. of the 25th Intl. Conf. on Computer Design*, pages 55–62, October 2007.