A Software-managed Approach to Die-stacked DRAM

Mark Oskin
AMD Research, University of Washington
mark.oskin@amd.com

Gabriel H. Loh
AMD Research, Advanced Micro Devices, Inc.
gabriel.loh@amd.com

***Abstract*—**
Advances in die-stacking (3D) technology have enabled the tight integration of significant quantities of DRAM with high-performance computation logic. How to integrate this technology into the overall architecture of a computing system is an open question. While much recent effort has focused on hardware-based techniques for using die-stacked memory (e.g., caching), in this paper we explore what it takes for a software-driven approach to be effective. First we consider exposing die-stacked DRAM directly to applications, relying on the *static* partitioning of allocations between fast on-chip and slow off-chip DRAM. We see only marginal benefits from this approach (9% speedup). Next, we explore OS-based page caches that *dynamically* partition application memory, but we find such approaches to be worse than not having stacked DRAM at all! We analyze the performance bottlenecks in OS page caches, and propose two simple techniques that make the OS approach viable. The first is a hardware-assisted TLB shoot-down, which is a more general mechanism that is valuable beyond stacked DRAM, and enables OS-managed page caches to achieve a 27% speedup; the second is a software-implemented prefetcher that extends classic hardware prefetching algorithms to the page level, leading to 39% speedup. With these simple and lightweight components, the OS page cache can provide 70% of the performance benefit that would be achievable with an ideal and unrealistic system where *all* of main memory is die-stacked. However, we also found that applications with poor locality (e.g., graph analyses) are not amenable to any page-caching schemes – whether hardware or software – and therefore we recommend that the system still provides APIs to the application layers to explicitly control die-stacked DRAM allocations.

***Keywords*-DRAM caching; die stacking; prefetching; memory; TLB**

I. Introduction

Recent advances in die-stacking (3D) technology have enabled the tight integration of significant quantities of DRAM with high-performance computation logic [1], [2]. Architects have focused on microarchitecture proposals for utilizing this resource as a cache for external off-chip DRAM [3], [4], [5], [6], [7]. Microarchitecture-only changes have the advantage of eliminating software adoption barriers, but they have the disadvantage of adding hardware complexity. A hardware-managed cache is only a small portion of the die-stacked DRAM design space, however, and in this work we consider an alternative perspective, which is what would it take to enable software to manage die-stacked DRAM?

Application-driven Allocations: The seemingly most straightforward software approach to using die-stacked DRAM is to expose its existence to applications with a conventional memory interface. This can be accomplished with adaptations of the existing NUMA allocation [8] facilities of modern operating systems. We call this an *application-driven* approach to die-stacked DRAM. We find that if applications could direct all of their memory requests to stacked DRAM, they would be 56% faster over a conventional system with non-stacked DRAM – the *memory wall* [9] still matters. Using all die-stacked DRAM is not reasonable, however, as it will constitute only a fraction of the total physical memory (20% in our model). Hence, applications must partition their allocations between fast and slow memory. Unfortunately, we find that for all but the most trivial of applications, it is not obvious which objects should be allocated into stacked or off-chip memory. The challenge confronting the software developer is a novel one: traditional reasoning about a NUMA system considers which threads access a chunk of data; here, the question is *how much locality* is in the access stream of all threads to this data? Feedback-directed compilation can assist matters, but ultimately even the optimal static partitioning of memory allocations leads to a meager speedup (9% over non-stacked DRAM). The reason is if an application accesses a memory region with high locality, then the existing processor caches serve most loads and stores. If a region has low locality, then it also tends to be large and unable to fit in die-stacked DRAM.

OS Page cache: If *static* partitioning of application data between on- and off-chip memory in software is insufficient, how does *dynamic* partitioning in software perform? The page table already maps virtual addresses to different physical memory pages. We propose to use the page fault handler to manage die-stacked DRAM *as if* it was a cache for off-chip DRAM. This is similar (in spirit) to how today's page fault handlers manage (off-chip) DRAM as a "cache" for virtual memory pages on disk. We call this approach an *OS-managed page cache*. Unfortunately, we find this approach does not work well. Performance is 48% *slower* with an OS-managed die-stacked DRAM cache than having no stacked DRAM at all.

Improving the OS-managed Page cache: On x86/x64, the reasons for this slow-down surprised us: the high cost of the OS-managed page cache is due to the high cost of TLB shoot-downs required on page table changes and not the cost of the processor trap and migration of data. To address this, we propose a simple architectural mechanism to selectively invalidate entries in the TLB of remote processors. This mechanism is a special inter-processor signal that invokes microcode on the receiving processor(s) that silently performs the TLB shoot-down and synchronizes with the sending processor, all without switching processor protection modes or invoking system code. This mechanism is useful for any page table change, thereby addressing a more general design problem with multicore processors. We find that with a hardware-assisted TLB shoot-down, using die-stacked DRAM as a software-managed page cache achieves a 27% speedup (about half of the possible benefit of the ideal all-stacked DRAM case). By comparison, a hardware-managed page cache achieves a 39% speedup.

Next we consider OS-directed prefetching, as prefetching in general is an effective technique to increase the hit rate of last level caches [5]. Critically, prefetchers exploit the *predictability*, not just the *locality*, inherent in the access (or miss) stream of a cache. While prior work has extensively studied prefetching at typical cacheline size granularities (32-64B), it is not entirely clear that a straightforward adaptation of prefetching will work with page caches where the line size is 4KB. We find there is indeed ample predictability in the miss-stream of a page cache at page-size granularities. We study both stride and correlation [10] Markov-model prefetching [11]. A software-based prefetcher increases the performance of our software page cache from 27% to 39% (70% of the ideal benefit).

The modest, yet observable, performance difference between all of these options suggests there is a rich design space that system architects can pick and choose solutions from for their target market. We explore each of these ideas in turn in this work. We begin, however, in the next section with background on die-stacked DRAM and a review of past architectural approaches in this design space.

II. Background

A. Die-Stacked Memory

Die-stacking technology enables multiple disparate silicon die to be packaged together with many low-latency, low-energy interconnects among the different die. This technology has been used to implement vertical stacks of memory, which can then be stacked on top of [12], [13] or next to ("2.5D stacking") [14] a processing chip. With 2.5D stacking, it is possible to incorporate multiple memory stacks into the same package.

Size: Engineering samples of the Hybrid Memory Cube (HMC) from Micron provide 2GB of capacity [1]. Projecting forward, if the process technology continues to shrink for another two generations (4$\times$ increase in density), and the number of stacked layers also increases by a factor of 2-4, this would provide an overall per-stack capacity of 16-32GB. Allowing for two or four stacks of DRAM on an interposer using 2.5D integration brings the total to 32-128GB of in-package, high-bandwidth memory (assuming two stacks with a doubling of layer counts, and four stacks with 4$\times$ layer counts, respectively). Some current servers already support hundreds of gigabytes of main memory. Assuming 512GB of off-package memory combined with 128GB of in-package die-stacked memory, the die-stacked DRAM would account for approximately 20% of the overall available DRAM. The exact fraction will naturally vary depending on how integration technology progresses and the required memory capacity of the target system.

Performance: Current servers typically employ DDR3 memory interfaces. A single channel of DDR3-1600 memory provides a peak bandwidth of 12.8GB/s. Server chips also support multiple memory channels; for example, an AMD Opteron$^{\text{TM}}$ 6300-series processor has four channels, for a total bandwidth of 51.2GB/s. Emerging die-stacked DRAM, however, supports significantly higher levels of bandwidth because the integration of the memory directly in the same package as the microprocessor avoids the conventional pin-count limits on both the memory and processor packages. As an example, the JEDEC High-Bandwidth Memory (HBM) standard [15] provides an interface consisting of eight independent 128-bit channels running at 500MHz DDR (1Gbps/pin). This provides 128GB/s per HBM stack; with two or four stacks, the total die-stacked memory bandwidth is 256GB/s or 512GB/s, respectively. Compared to the conventional off-package DDR3 interface, the die-stacked DRAM can provide a 4-8$\times$ improvement in memory bandwidth.

B. Related work

Die-stacked DRAM caches: Past work has proposed using die-stacked DRAM as a software-transparent, hardware-managed cache at conventional cacheline [3], [4], [7] and page granularities [5], [6], [16]. Such approaches minimize software changes, but they also come with intrinsic implementation challenges. If a typical cacheline size is used (e.g., 64B), then the area devoted to tag storage will be egregious. Various solutions that trade-off performance and flexibility have been proposed [5], [6] to mitigate this. If large (page-size) cachelines are used, then off chip bandwidth is not always efficiently used. Thus mechanisms that manage blocks at the page-granularity but fill them at a finer one have been proposed [5]. Finally, because on-chip DRAM is only slightly faster than off-chip DRAM, the added latency of checking tags on a miss can be costly, and thus mechanisms to predict a hit or miss [17], [18] have been developed.

Software approaches are desirable from a hardware implementation standpoint as changes required of the microprocessor are minimal: the processor memory complex (e.g., Northbridge) must check the address of an incoming request, and route it to either fast or slow DRAM. This is not fundamentally different than what occurs now, routing address-to-{bank,channel,NUMA node} in existing hardware. We believe that this is the first work that explores the effectiveness of having the software directly manage a die-stacked DRAM cache, rather than relying on more complex hardware-management implementations.

NUMA & DSM: There has been extensive research on optimizing memory allocations on NUMA systems [19], [20], [21], [22], [23], [24], [25]. Prior work, however, has focused on systems where there is an opportunity to move computation relative to memory. The die-stacked "NUMA" system we consider here has no such degree of freedom; all processors are equally close to the stacked DRAM (and equally far from off-chip memory). Solutions therefore must focus on partitioning allocations based on the access patterns of all threads in the computation, instead of focusing on matching the access patterns of a particular thread to a given memory store. Ramos *et al.* considered an intelligent memory controller working in conjunction with an operating system that dynamically partitioned memory pages between phase-change memory (PCM) and DRAM [26]. While related in spirit, the design (dynamic partitioning versus caching), focus (robustness versus performance and easy integration into the x86/x64 platform), and technology (PCM versus die-stacked DRAM) are distinct.

Software distributed shared memory (DSM) systems have been studied extensively [27], [28], [29], [30], [31]. The similarity between our work on software-managed die-stacked DRAM caches and prior DSM efforts is that both rely on software control of the page-fault handler implemented entirely in the operating system or partially in user-mode [32]. DSM systems aimed to provide a global shared address space with sequential [27] or more relaxed [28], [29], [30] consistency model semantics. In contrast, our goal is to migrate pages efficiently between on- and off-chip DRAM of a single-system. Nevertheless, the innovations described in this paper on improving the TLB shoot-down cost would be applicable to DSM systems. Finally, while not studied in our present work, utilizing different page sizes [31] may prove useful.

TLB Shoot-down: In this work we propose and evaluate a hardware-assisted TLB shoot-down mechanism. The cost of TLB shoot-downs and ideas to improve it have been explored at length by researchers [33], [34], [35], [36], [37], [38]. These previous designs attack the cost of TLB consistency from different directions: shared TLB structures, coherence between the page table and the TLB, and filtering of shoot-down requests. The ARM11mp core supports a shared-TLB structure, where invalidations flush the micro-TLBs contained in each core [39]. Our work focuses on a simple and selective mechanism that is implementable via microcode patching of existing x86/x64 processors. The creativity in our design stems from threading the complex path through the legacy x86/x64 architecture to devise a method that requires little practical implementation effort.

Prefetching: In this work we consider stride [40], [41], [42] and Markov [11] prefetching schemes, but extended to page granularities. We also explore prefetching strategies that rely on separate prefetch caches [43] and those that do not. These ideas have a long history of research in computer architecture. The innovative component of our work is the *application* of these ideas to page-level caching and the exploration of whether prefetching should be done in hardware or software.

C. Baseline Hardware

To ground our work in realistic data points, we consider a processor design similar to the AMD Opteron 6168 processor (1.9Ghz, 512KB caches, 64B cache lines). We also assume conventional DDR3 channels to off-chip memory, and the previously described in-package, high-bandwidth interface to stacked DRAM.

Our target environment is large server-class systems, and as such we do not expect to be able to stack all of system memory on chip. In fact, we expect only about 20% of total system DRAM will be die-stacked with the remainder being off-chip, accessible over conventional DDR interfaces. Because of this configuration, main memory will appear to the processor to have two different speeds. In this paper we refer to the in-package, die-stacked DRAM as "fast" and the conventional, off-package, DDR-based DRAM as "slow."

III. Application-driven Allocations

The simplest solution for managing two speeds of DRAM does not involve hardware at all, but only software. An application explicitly chooses to allocate objects in slow or fast DRAM. While implementing this approach it became apparent there are at least three challenges facing a software developer:

Challenge #1: Modifying legacy source code to use these speed-specific allocation functions is *not* a trivial task. At each call site, the software developer must reason about the *size*, *significance*, and *locality* of accesses to the memory being allocated. Manually porting large code bases with thousands of allocation call sites does not seem practical.

Challenge #2: Many applications centralize or intercept allocation functions. They do this for at least two reasons that we found: (1) to add debugging code, such as routines that track allocations and frees; or (2) to provide different allocation semantics, such as allocation pools.

Challenge #3: Allocations are often buried deep in library calls. For example, applications that use the *fftw* [44] library invoke that library to allocate memory that will be used to carry out the FFT. Whether that memory should be allocated on fast or slow DRAM is really a joint question for both the caller and callee to decide. This is not a trivial interface for a software engineer to design.

Program	L1 accesses (L+S)	LLC loads	DRAM accesses	L1 load-miss rate	LLC load-miss rate	Footprint
CoMD	112141M	1749M	201M	.72%	11.49%	112MB
LU-c	9060M	58M	6M	.34%	10.34%	32MB
LU-nc	9487M	1157M	72M	6.37%	6.22%	32MB
LULESH	86223M	4873M	650M	1.79%	13.33%	19MB
BARNES	28071M	482M	31M	1.44%	6.43%	1303MB
OCEAN	8604M	722M	37M	.90%	5.12%	887MB
FMM	28125M	271M	111M	.73%	40.95%	528MB
FFT	3910M	184M	21M	1.70%	11.41%	768MB
g500csr	7884M	1679M	911M	21.84%	54.25%	139MB
miniFE	-	-	-	-	-	370MB

Table I: **Test programs:** The following benchmarks were used to gauge performance. In addition cache behavior and memory footprint is depicted. Cache behavior for miniFE is omitted due to limitations in our ability to collect data on MPI applications.

Methodology: To study whether application-directed allocation functions are a viable approach, we created a modeling infrastructure that takes an existing hardware platform and, through the introduction of contention, creates two different speeds of DRAM. We use a 48-core four-socket (12 cores/socket) system board to mimic a one-socket 12-core system. Each socket has two DDR3 DRAM memory channels (8 channels total). Our emulator dedicates the first socket for execution of the application and the hosting of "fast" memory. The second socket is dedicated to host the "slow" memory. The memory on the third and fourth socket is not used, but the processors on those sockets are configured to introduce contention on the memory channels on the second socket. For our experiments, we calibrated the level of contention such that the emulated stacked DRAM is 4.5× faster compared to the emulated slow memory (or in reality, the slow memory has been slowed down by 4.5×) for sequential accesses. This provides a similar memory bandwidth performance ratio of a 51.2GB/s off-chip memory system compared to 256GB/s of die-stacked DRAM.

Equipped with a system with distinct memory regions with different memory performance characteristics, we can then use the *libnuma* [8] support within Linux to specifically place allocations on "fast" memory as needed for each experiment. We replaced all memory allocation functions from *libc* with our own, which can be configured on a per-call site basis to direct allocations to fast or slow memory. In addition, we modified the source code of our applications (Table I), as needed, to remove any application-level memory debuggers or intercepts (see Challenge #2 above).

Selecting Fast vs. Slow Allocations: While Challenges #2 and #3 are only fixable through invasive source code changes, tools can potentially help automate the process of tackling Challenge #1. We developed a feedback-directed compilation system that partitions malloc calls (by static call site) with a hill-climbing algorithm between fast and slow memory. The tool begins by forcing all allocations (larger than 32KB) to use slow off-chip DRAM. The tool then selectively measures the performance of changing a single allocation to use fast on-chip memory. After trying each individual allocation, the one with the greatest performance increase for the lowest memory footprint is chosen to be placed in fast on-chip DRAM. This process is then repeated until all fast memory is allocated. This hill-climbing strategy is quadratic in the number of memory allocations. We compared this strategy to an exhaustive search (where tractable) and found placement results to be qualitatively similar. Our experiments provide the best chance for static, application-driven partitioning to work: we use the same input sets to train and measure final performance, and running an application a quadratic number of iterations is likely impractical in many real scenarios.

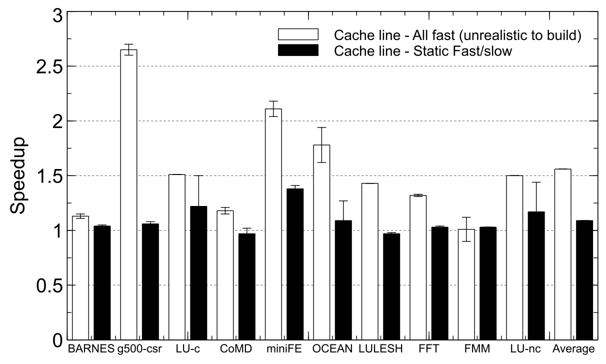


Figure 1: **Baseline performance:** speedup obtained from (unrealistically) storing all program data in die-stacked DRAM compared to the speedup from an optimal (found via hill-climbing) static partitioning between die-stacked and off-chip DRAM.

Results: Figure 1 shows the results of our experiment. The results depict the average of 20 runs per application. For each application, the first bar shows the ideal speedup if all program data are stored in on-chip, die-stacked DRAM, normalized to the baseline where everything is in off-chip DRAM (56% on average). The second bar shows the speedup achieved when using our static partitioning (via near-optimal hill climbing) of allocations. Overall the average performance benefit of static partitioning is 9%. This is not entirely surprising, as at most only 20% of an application's memory can be statically placed in fast memory.

Looking at individual applications, we see some with little performance difference between ideal and near-optimal (BARNES, CoMD, FMM), others that see moderate differences (LU-c, LULESH, FFT, LU-nc, OCEAN), and still others that see a significant difference (g500-csr, miniFE). In general, in order for an application to see a performance impact from a faster DRAM subsystem, it must access it frequently. Table I shows measured cache behavior for our test applications (from hardware performance counters). The data show why g500-csr performs poorly: both the L1 and LLC cache hit rates are poor. Similarly, the data explain why LU-nc sees a moderate impact. Applications such as LULESH and FFT lie in the middle as well, but due to a coupling of reasonably high L1 hit rates and reasonably low LLC hit rates. Finally, applications such as CoMD and FMM have extremely high L1 hit rates, indicating most of their accesses are captured by the L1 cache.

Overall, the conclusion here is if a region is accessed frequently it tends to end up in the processor cache anyway. Thus, statically partitioning memory regions between fast and slow memory is *only effective if* a small region of a program's memory, exceeding the size of the processor cache yet being smaller than the size of die-stacked DRAM, is accessed with poor locality. This is not the common case for the applications evaluated, and therefore the meager performance benefits of manually partitioning the allocations at source-code level likely do not justify the high level of programmer effort.

IV. OS-defined Page Caching

The previous section explored whether *statically* partitioning application memory between fast and slow memory was an effective idea. We found that in general, it is not. Another option is to *dynamically* partition memory. The classic architectural approach to do so is to use a cache. Researchers have previously explored using die-stacked DRAM as a hardware cache for off-chip DRAM [45], [3], [4], [5], [6], [16]. But if the "cache-line" size is the same size as a virtual memory page (4KB), a tantalizing software-only approach to building a cache exists. The existing hardware support for virtual address translation can be dual-purposed to build a page cache. To do this, the OS initially maps application memory to the slower off-chip memory, but access is disabled via the page table. The OS then maintains a set of DRAM pages in fast memory to act as a page cache. As the application executes, it will generate page-fault exceptions when a request is not in the die-stacked DRAM cache. The OS then copies the data from slow to fast memory, and adjusts the application's page table entry accordingly. Just as with a hardware cache, the OS must, in time, remove a page from fast memory. To do so, access to it must be disabled to the application, and then the page, if dirty, must be copied back to slow memory. While this minimizes the hardware support required to implement a stacked-DRAM cache, in such a design there are several additional costs:

Processor fault: When an application accesses memory not addressable from its page table, a fault occurs. This fault is a precise interrupt, and because the access must be assured to be non-speculative, the processor pipeline is drained. The processor privilege mode is switched (if execution was in user-mode), and a trap handler is invoked. If a *user-mode* software trap handler is used, the kernel trap handler reflects the fault (via a SIGSEGV for POSIX systems) back to the application, thereby causing yet another privilege switch.

Page migration: An all-software cache requires that the *entire* page of data be migrated from off-chip to on-chip, and, in the common case, an evicted page of data be migrated from on- to off-chip. Because of the time (and energy), prior hardware approaches considered allocating cache space at the page granularity, but filling it with data at a finer level (64B) [5]. A software page cache does not permit such sectored approaches [46], which implies that a write to even a single byte causes a write-back of a full page upon eviction.

TLB shoot-down: The OS-defined cache manipulates the page table to reflect the fact that an application's accesses to a region of memory should be directed at the copy of that data stored in fast on-chip memory. In addition, when the cache evicts a page from on-chip memory, the page table must be updated to disable access to it. On x86/x64 systems, changes to the page table are *not* kept coherent with the translation look-aside buffers (TLB) of the processors. As such, when the page table is changed by the OS, the TLBs potentially contain stale entries. It is up to the OS to "flush" these stale entries via a software routine executed on each processor that may contain them. This process is known as a "TLB shoot-down".

To study the effectiveness of an OS-based page cache, we extended our platform to functionally emulate the page cache activity; for the purposes of rapid design-space exploration, we also use this facility to collect traces for faster off-line analysis and performance modeling. The trace generator is compiled into applications and provides a *malloc_paged* call to the user application. Memory that is allocated on a paged region is initially *not* mapped into user space. When the application first touches a paged memory region, a SIGSEGV is generated and handled by the user process. The signal handler uses the Linux API calls to map the faulting memory into the process address space. It also maps *out* a paged memory region on a page cache eviction. In this way only a finite amount of paged memory is accessible to the user application at any one point in time. Ultimately, this tool generates a trace of page faults that reflect the "misses" to the page cache. Using the trace, we can use offline analysis to determine the performance impact of, for example, varying the cache-miss/page-fault overheads. Because our page cache is implemented in user-mode we cannot manipulate the access and dirty bits within the page table; consequently, a first-in/first-out page replacement policy is utilized.

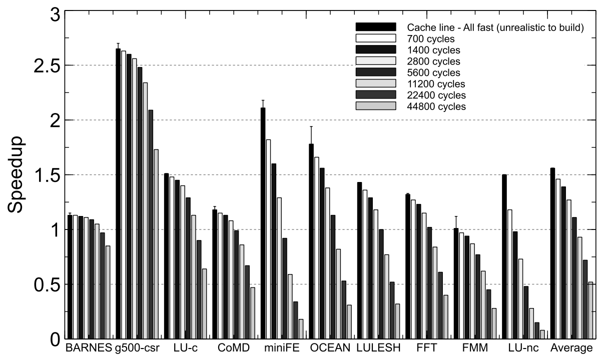


Figure 2: **Baseline performance:** speedup obtained from using an OS-managed cache; results include a sweep over various fault-times. Idealized performance is also shown for comparison.

Results: Figure 2 shows application performance when the OS page cache is used. The graph consists of seven sets of bars. Within each set, each bar is the speedup with a page cache assuming a given cycle-cost for a page cache miss. Depending on the fault overhead (typical measured values are 13K-55K cycles), the OS-managed page cache suffers a 7-48% slow-down compared to *not using stacked DRAM at all*. In comparison, a hardware-managed page cache would achieve approximately a 39% speedup for these applications (a hardware cache would have a cache miss overhead in the range of ∼700-1,400 cycles, corresponding to the left-most two bars per set). These results show that an OS-based page cache, at least without any additional help, is not likely to provide an attractive solution. In the next section, we take a closer look at the sources of overhead in the OS-based page cache, and then propose techniques to address these issues and ultimately make the OS-based approach much more practical.

V. Fixing the OS-based Page Cache

The previous section demonstrated that an OS-driven page cache seems like a poor implementation choice. We find that the primary culprit is the trap overhead related to TLB shoot-downs. We tackle this in two ways: (1) reduce the cost of the trap and cache management routines, and (2) reduce the frequency with which the traps occur. In this section we explore each of these.

A. Reducing Trap Overhead

Figure 3 shows the cost of a page cache miss broken up into its constituent parts. Overall the cost of servicing a page cache miss is ∼13K-55K cycles, or roughly 20-80× the cost of *just* moving the data. But it is worthwhile to explore where the cycles are spent in the miss handler. There are four components: the first is the data migration to and from fast and slow memory; the second is the added cycles if a user-mode instead of kernel-mode trap handler is used. This reflects the measured cost of a fault in user space, the

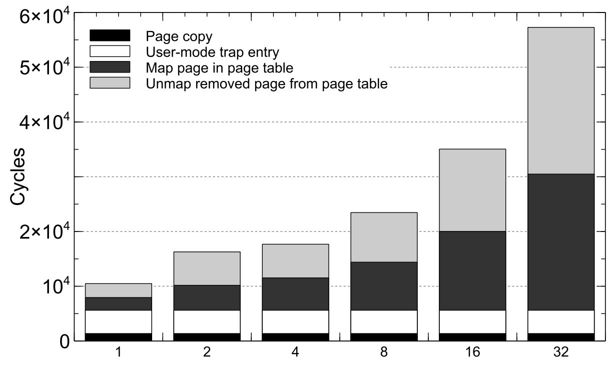


Figure 3: **Trap overhead:** In this graph, the runtime cost of handling a trap for the page cache is broken out into its constituent parts. Each bar is for a given number of user-mode threads in the application being measured. For each bar, four components are broken out: (a) the cost of the actual data migration from off-chip to on-package DRAM; (b) the added overhead if the trap is handled in user space instead of kernel space; (c and d) the cost of the Linux kernel map and unmap functions for adjusting the page table.

entry into the kernel, scheduling, re-entry to user space to handle the fault, leaving user space to re-enter the kernel, scheduling, and then re-entering the user mode application to continue execution. In total, four user/kernel mode switches are required by our handler; the third and fourth are the costs of adjusting the page tables to map and unmap memory from the page cache. These components also require two to four user/kernel mode switches if a user-mode handler is employed. We measured this component to be highly dependent on the number of threads in the user application on our hardware platform.

The data in Figure 3 illustrate a surprising fact about the page cache with a user-mode handler: the dominant cost is not the "user-mode" aspect of it, but rather the kernel implementation of the page table manipulations. There are two types of page table changes required to implement a page cache. The first is an unmap operation, that removes access to the physical page from the processor. The second is a map operation making it accessible. We have found that in both cases, the Linux kernel dispatches an inter-processor interrupt (IPI) to all processors executing with the page table being manipulated in order to synchronize page table changes, even though in the map case the shoot-down is not actually required. This makes performance dependent on the number of active threads in an application, and moreover this TLB shoot-down process is the dominant cost in the software-based page cache.

Prior work [33] traced the cost of TLB shoot-downs. In that work, the authors found the average cost of shoot-down grows with thread count, starting at ∼2K cycles with two threads and rising to ∼11K with 16 threads. We wrote a microbenchmark to explore TLB shoot-down cost in Linux and corroborated their results, finding a further jump to ∼25K cycles with 32 threads. There is no fundamental reason why a TLB shoot-down needs to be this expensive. Not only does the high cost of a TLB shoot-down make a software-defined DRAM cache prohibitively expensive, it also can have a major performance impact on systems without die-stacked DRAM. Prior work has shown that TLB shoot-downs cost up to 10-20% of system performance [47].

How TLB shoot-downs work today: To understand how to accelerate the TLB shoot-down process, it is worthwhile to examine how it currently works in software. The OS issues a TLB shoot-down when it changes a page table[1]. The OS does this because the processor does not keep its TLBs coherent with the page table in memory. Changes to the page table necessitate flushing the TLB of the older (stale) entry/entries. On x86/x64 processors, the INVLPG instruction flushes the TLB of a specific mapping, while certain changes to the CR3 and CR4 registers flush the TLB of all local and/or global (shared across page tables) entries. Invoking this flushing process on a single processor core clears the TLB of the relevant entries *on that core only*. Hence the OS must, in software, issue the necessary TLB flushing instructions *on each core of the system* where there potentially is a stale TLB entry. This process is known as a TLB shoot-down.

To perform a TLB shoot-down, the OS notes which actions to perform in a data-structure stored in shared memory. It then initiates an IPI to all other processors in the system in order to invoke an operating system routine to conduct TLB invalidations. An obvious optimization, and one implemented in Linux, is to only send an IPI to those processors that have or potentially have the stale mappings. This is implemented by tracking which page tables have been set for each processor. Changes to the page table are typically synchronous with program execution. A critical fact to note is that because of this, the OS-IPI path for a TLB shoot-down also includes synchronization with the processor that dispatched the shoot-down.

The Linux TLB shoot-down pseudo-code is shown in Figure 4. Villavieja *et al.* found the dispatch loop contributes the majority of the time cost [33]. We found the source appears to be the high cost of interfacing with the Advanced Programmable Interrupt Controller (APIC) [49]. The APIC, first introduced as an external chip (the 82489DX) in the 486 series of processors, lives on in present day x64 multicore processors largely unchanged except for being integrated

```
tlb_shootdown(void *addr) {
    mailbox.addr = addr;
    mailbox.count = affected_processors.size();
    for (processor in affected_processors) {
        wait_for_APIC_to_be_idle();
        send_IPI(shootdown_entry);
    }
    shootdown_entry();
    block_until_zero(mailbox.count);
}

shootdown_entry() {
    invtlb(mailbox.addr);
    atomic_decrement(&mailbox.count);
    send_eoi_to_local_apic();
}
```

Figure 4: Pseudo-code of Linux TLB shoot-down code

on-die. The TLB shoot-down routine interacts with the APIC in two calls: waiting for the idle bit in the interrupt dispatch register, and then initiating the IPI. The idle bit has already been deprecated by the X2 APIC [50] (presenting an always-idle interface), although it exists in the legacy (and more widely used) APIC. Once the APIC is idle, the kernel then initiates an IPI dispatch to a specific processor. The current APIC hardware interface is fairly rigid about broadcasting to either all processors or only to preconfigured processor groups. Because of this rigidity, Linux converts any (selective) broadcast IPI into a loop of individual IPIs on the legacy APIC, and a loop across processor clusters (with per-processor selectivity) on the X2 APIC. Once the IPI is received, the receiving processor switches out of user mode (if needed), switches stacks (if needed and configured), saves registers, and executes a kernel mode handler. This handler reads which invalidations it must perform from shared memory, performs the actual invalidation, performs a synchronization operation via shared memory with the sending processor, sends an end-of-interrupt (EOI) signal to its local APIC, and then returns, which causes the processor to restore registers, switch stacks (if needed), and (if needed) returns to user mode.

Hardware-assisted TLB shoot-down: As previously discussed, many researchers have proposed hardware designs to improve the cost of TLB coherence [51], [33], [34], [35], [36], [37], [38]. These designs all have their merits, particularly in architectures designed from a clean slate. Our goal is slightly different, which is to accelerate the TLB shoot-down process with minimal changes that are compatible with the x86/x64 ecosystem. There are many practical legacy architecture and microarchitecture factors to consider. We propose a simple change involving two new architectural features. The first is a special form of IPI, which we call a REMOTE_INVLPG, and the second is a microcode change that receives this special IPI and issues a TLB shoot-down process entirely in microcode without necessitating any OS interaction. Because the APIC on x86/x64 processors is implemented as RTL, changes to that portion of the architecture are more complex to implement than microcode changes. Hence, in our proposal we leave the APIC untouched. Because of this, the majority

[1]Certain changes to the page table, specifically those that *enhance* the permissions (e.g., read-only to read/write), don't necessarily require an immediate TLB shoot down. The processor with the outdated TLB mapping will issue a page fault and the page fault handler can inspect the mapping to determine that the fault is caused by a stale TLB entry and not by a true fault in program execution. This is known as *lazy TLB updating* [48]. Past work has shown this to be a useful optimization, but our own measurements show that the Linux kernel does not utilize it in all cases where it could have.

```
struct tlb_shootdown {
    u_int64_t    ipi_count;
    u_int64_t    processor_mask_size;
    u_int64_t    processor_mask[];
    int64_t      entry_count;
    u_int64_t    entries[];
    };
```

Figure 5: Data structure used to communicate TLB shoot-down information/parameters.

of information communicated for the TLB shoot-down is sent via shared memory with a data structure shown in Figure 5.

When the OS needs to issue a TLB shoot-down process, it atomically acquires access to this structure by setting the IPI_COUNT variable to the number of processors whose TLBs need to be invalidated, plus one using the compare-and-swap instruction (the structure is logically not in use when IPI_COUNT is equal to zero). The OS configures the structure as follows: the ENTRY_COUNT field is set to the number of values in the ENTRIES array, with two special ENTRY_COUNT values, "all local" and "all global", being provided. These special values indicate to the receiving processor that the ENTRIES array should be ignored and instead all local or all global entries should be invalidated; ENTRIES is a list of addresses to be invalidated; finally, the PROCESSOR_MASK and PROCESSOR_MASK_SIZE fields provide a bitmask representing which processors should perform the TLB invalidations. These last two fields are not necessarily required if more invasive changes to the APIC RTL design and interface are carried out. Once the invalidation structure is filled in by the dispatching processor, the OS dispatches a REMOTE_INVLPG operation via a broadcast interrupt sent via the APIC.

The REMOTE_INVLPG signal is really nothing more than an interrupt configured via a model specific register (MSR) to be the TLB shoot-down interrupt. When a processor receives an interrupt dispatch request from its local APIC, it checks the MSR holding the interrupt number for the REMOTE_INVLPG routine. If a match is not found, it invokes the existing interrupt microcode. If a match is found, it inspects the TLB_SHOOTDOWN structure in microcode. The location of this structure is pre-configured by the OS in each processor via an MSR register. If the processor is not in the PROCESSOR_MASK field, it sends an end-of-interrupt (EOI) signal to its local APIC and carries on with what it was previously doing. While this does cause all processors to spend a few extra cycles in their front-end processing an interrupt in microcode, the performance impact is minimal. The main cost is the cacheline holding the PROCESSOR_MASK field must be migrated to the L1 cache of all processors (read-only for processors not in the receive set). If the processor is in the PROCESSOR_MASK field, it performs the requested TLB invalidate operations, atomically decrements the IPI_COUNT value, and sends an EOI signal to the local APIC on completion. The receiving processor then continues to execute normally. Note that no user/kernel mode or stack switch is required as no actual user or kernel code is executed to complete the operation.

The processor that initiates the TLB shoot-down operation can determine when all receiving processors have completed their TLB invalidations by monitoring the IPI_COUNT variable. When the value goes to one (1), the operation is complete. It can then set the value IPI_COUNT field to 0 which releases the lock on the structure.

To understand the performance impact of this improved TLB shoot-down mechanism, we measured three separate operations that represent the constituent parts. The first is the cost of acquiring access to the shared data structure and filling in the relevant entries. We found this cost to be $\sim$377 cycles on average when 16 processors contend for this resource (an unrealistically high amount of contention in our system, but sufficient to bound the cost). The second is the cost of sending and processing the REMOTE_INVLPG operation. Our measurements for dispatching an IPI are in line with past measurements [33] and require $\sim$1,500 cycles. The third component is the synchronization cost, which we found to be approximately $\sim$300 cycles when 16 processors are issued the TLB shoot-down routine. In total we expect the microcode-assisted TLB shoot-down to require less than $\sim$2,500 cycles, with the dominant cost being interfacing with the legacy APIC hardware (which we imagine can be further accelerated, but is not something we consider in this work). Note this value corroborates fairly closely with our measurements for a two processor end-to-end remote TLB shoot-down request ($\sim$2,200 cycles) as measured on our test system. It also corroborates with the same two processor end-to-end remote TLB process from prior work [33].

Results: When a fast micro-code assisted TLB shoot-down is used, we find that an OS-managed page cache is a viable idea. What used to be a 1.5-5.8$\times$ slowdown is now a 27% speedup (corresponding to the "2800-cycle, no prefetching" data in Figure 2). This makes the OS-managed page cache faster than the static application-driven approach, and provides nearly half (48%) of the performance gains available with an unrealistic all-fast memory system.

B. Prefetching

Next we consider the question, what if prefetching is employed to reduce the frequency of traps/TLB shoot-downs? When servicing a page cache miss, in addition to copying the requested page to the fast memory, the OS can attempt to prefetch additional pages, and then update all of the page table entries and shoot-down all affected TLBs at once. However, it is not entirely obvious that prefetching will be effective for caches with page-sized lines. For example, strides within a page would not be visible to the software (only the first access to the page will generate an OS-visible miss), and therefore the strides must exhibit some repetitive pattern at the inter-page level. While stride prefetching exploits the predictability of accesses, it can also be effective simply because of more intrinsic spatial locality existing in the access stream than the cache line size is naturally exploiting. We consider two different types of prefetchers: stride [40], [41], [42] and Markov-model [11] based in this work.

Methodology: We extended our trace-based page cache model to study stride and Markov-model prefetching. This model is configurable and enables us to study a variety of stride and Markov-model parameters. Moreover, it models prefetching carried out in software or in hardware. Finite DRAM bandwidth and true latencies are modeled, as well as the cost to generate and manage all prefetch data structures. For stride prefetching, we consider a finite number of strides, maintained in an LRU fashion. Prefetch requests are generated by applying strides to elements in a history buffer of past page cache misses. For each stride and element in the history buffer, multiple prefetch requests are generated in order to aggressively prefetch pages many stride lengths away. For each prefetch request generated, a priority is assigned, with nearer strides receiving higher priority.

The Markov-model prefetcher is configurable in terms of number of states, edges, and edges per state. Markov prefetches are generated from the current state (the most recent miss in the page cache), and the outbound edges, which represent different prefetch request possibilities, are considered in most recently followed order (i.e., which misses have most recently occurred immediately after the currently missed page). Because Markov states (unlike strides) are *tagged*, if a match is found in the model, we favor prefetching the outgoing edges in the model over distant stride prefetches.

Overall results: The data in Figures 1 and 6 suggest that different applications have more or less locality and predictability. For example, because BARNES and FMM perform well with all application memory in slower off-chip DRAM (Figure 1), the overall memory reference pattern must have high locality, as the on-chip processor caches are servicing most memory references. On the other hand, g500-csr has poor locality in its raw memory reference stream. To differentiate applications by their predictability/prefetchability, we consider the relative runtimes of the page cache with and without using a prefetcher (Figure 6). What we see is that several applications (OCEAN, LU-c, miniFE, g500-csr) see a large benefit from prefetching, while other applications (BARNES, FMM, FFT, LU-nc) see less of a benefit. Overall, prefetching at page granularity boosts the performance of an OS page cache from 27% to 39% speedup.

C. Implementation Considerations

It is straightforward to implement a page cache in software (performance considerations aside), but not so with prefetching. While the decision logic about *what* to prefetch

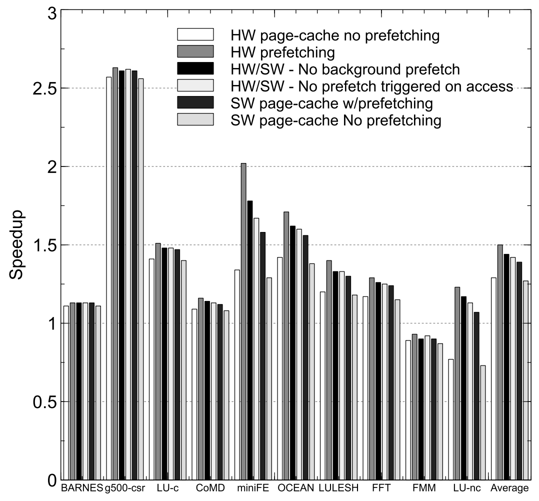


Figure 6: **Prefetching HW/SW Interface:** the impact of prefetching all in hardware, all in software, or with hardware-assisted software routines.

is easily implemented in software, exactly when to issue a prefetch and when to manipulate the application's page table to reflect the completion of a prefetch is unclear. A variety of implementation options exist, from an all-software approach that prefetches pages entirely in the fault handler that manages the page cache, or an all-hardware approach that observes the page cache accesses and works in concert with the software (or hardware) page cache routine to insert prefetched pages. To understand the efficacy of any of these options, we consider two key design considerations that transcend any specific implementation: (1) How critical is it that the prefetches occur outside the critical path of the page cache miss handler routine? (2) How important is it to trigger additional prefetches when the processor accesses a prefetched page? Figure 6 depicts the result when a stride prefetcher is used ($history = 1, strides = 4, depth = 1$) with different combinations of hardware and software implementation options. For each application, performance is shown relative to a page cache design that has no prefetching.

How important is asynchronous prefetching? As expected, in all cases, a hardware-only prefetcher proves the best performing option. When we disable background prefetching and force the software handler to complete all prefetches prior to continuing program execution, we see that performance deceases slightly (by 4%). While this is true for a OS page cache trap handler that uses the fast-TLB shoot-down mechanism, if present-day unmodified hardware is used, then there are some exceptions to this general performance trend. For instance, with only present-day hardware, LU-nc sees a performance *increase* from disabling asynchronous prefetches. The reason is it does very little computation per memory access. If the fault-handler dispatches prefetches to be completed asynchronously with application execution, they fail to complete in a timely manner [52]. In our HW/SW model, another software fault is triggered to wait for the prefetch to complete. The cost of this second fault is *more* than simply completing the prefetches synchronously within the first fault handler.

How important are prefetches triggered on a successful prefetch? When we disable the generation of additional prefetches on the access of a page that was successfully prefetched – as would happen if hardware wasn't tracking the prefetched pages – we see that performance is uniformly decreased for all applications (by 5% on average). The applications impacted most significantly are miniFE, OCEAN, LULESH, FFT, and LU-nc. These applications decease the prefetching accuracy when depth increases (Figure 7(a)) and in most cases when the number of strides supported decreases (Figure 7(c)). Consider a straightforward access stream of $\{1, 2, 3, 4, 5, 6, ...\}$. If the stride prefetcher does not see the successful access of its $stride = 1$ prefetch, its reduced information will enable it to see only $\{1, 3, 5, ...\}$. The prefetcher will then configure itself to fetch $stride = 1$ and $stride = 2$, because in our explorations we enable up to four different strides to be considered simultaneously. This continues for all four strides. The net effect of the reduced information is the depth is artificially increased and the number of strides considered is artificially decreased. As noted earlier, this is precisely the situation where these applications exhibit a less accurate prefetcher.

Finally, we consider when an all-software prefetcher is used. Prefetches must be generated and completed entirely within the software fault handler and the prefetcher is not aware of the additional information about whether a prefetch was actually used. Prefetches are inserted directly into the main page cache, potentially causing pollution. This achieves a 39% speedup overall. By comparison, a middle ground, a software page cache coupled with a hardware-prefetcher is only modestly faster (44% speedup). Finally, an all-hardware based page cache and prefetcher optimized to reduce wasted off-chip DRAM bandwidth achieves a 50% speedup. Overall, we note that a simple software-only prefetcher achieves $\sim$88% of the performance of a hardware approach and within $\sim$70% of an idealized one.

D. Prefetcher Parameters

In the next few paragraphs we explore the design space of stride and Markov-model prefetchers to understand what types of prefetching works well for a page cache.

Prefetch Cache Size: We found the single most important determiner of prefetching performance to be the size of the buffer used to hold prefetches (Figure 7(d)). In our prefetching model we do not place prefetches in the main page cache, but instead hold them separate region of on-package DRAM. As expected, applications with high predictability see little benefit from an increased cache size (BARNES, OCEAN, LU-c, miniFE). Applications with poor predictability (FMM, LU-nc, FFT) benefit from an increase in prefetch-cache size, allowing the prefetcher to more aggressively prefetch. The g500-csr application sits between these two extremes, being at the top end of applications that benefit from an increased prefetch-cache size, but also having relatively high predictability.

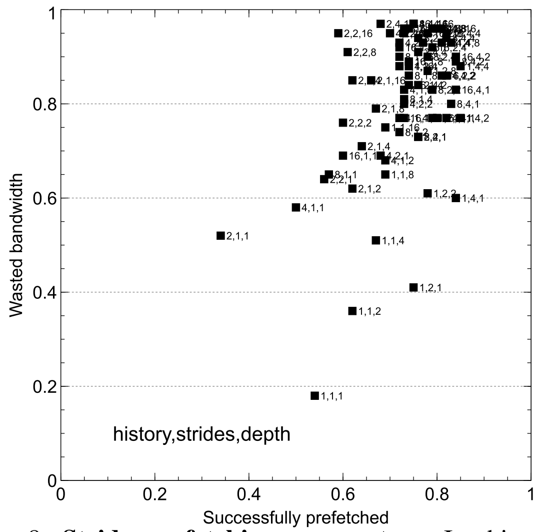
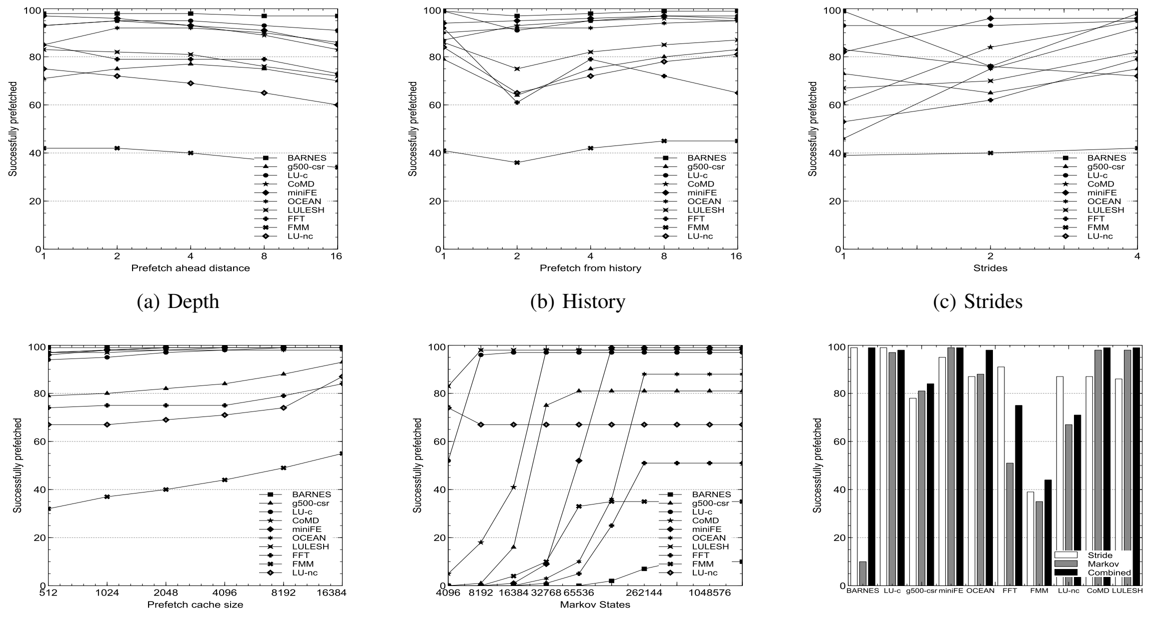


Figure 8: **Stride prefetching parameters:** In this graph we depict the success of a prefetcher versus the fraction of wasted DRAM bandwidth from useless prefetches.

Stride prefetch parameters: Figure 7(a,b,c) shows prefetch accuracy versus various configuration parameters for the stride prefetcher. The results in Figure 7(a) show that stride prefetching gains no benefit from looking too far ahead. In fact, the added pollution to the prefetching cache slightly decreases prefetching accuracy. Figure 7(b) shows that basing prefetches on anything but the last page cache miss is not effective. Figure 7(c) shows that four strides (utilized in an LRU fashion) is better than one or two.

Markov prefetching: Figure 7(e) shows that Markov prefetching is effective only if a relatively large (256K state) prefetcher is utilized. Additional states beyond this add little to the prefetch accuracy. We also explored designs that followed deep within the chain of Markov-model links. We did not find a viable approach to using/prioritizing these requests. The data shown come from utilizing Markov states that are only one hop away from the present state. Figure 7(f) compares stride, Markov, and combined stride+Markov prefetching. Overall, the data show that stride prefetching is more effective than Markov prefetching. For most applications, Markov prefetching adds a performance boost, and at least for some applications (notably FFT) Markov prefetching leads to a performance decrease. This is likely due to the generation of spurious prefetches polluting the prefetch cache, as Figure 7(d) illustrates that FFT is extremely sensitive to prefetch cache size.

Stride-prefetching design space: Figure 8 shows the entire design space of stride-prefetchers we explored com-



(a) Depth (b) History (c) Strides

(d) Prefetch-cache size (e) Markov states (f) Prefetch policy

Figure 7: **Prefetching parameters:** In these charts we depict performance with varying stride prefetch parameters. *Depth* refers to how far in advance (in unit strides) a page address should be prefetched. *History* is the number of past addresses to use to base prefetches from. *Stride* is the number of strides to maintain (LRU replacement). *Prefetch-cache size* is the size (in pages) of the prefetch cache. *Markov states* is the size of the Markov predictor (in states). *Prefetch policy* compares stride, Markov and combined stride+Markov prefetching policies

paring successful (useful) prefetches against prefetches that had no value and simply wasted DRAM bandwidth. The design points along the lower convex hull represent the Pareto frontier. Inner points waste bandwidth unnecessarily. What we observe is not too surprising: prefetches based off of only the last miss ($history = 1$) and that do not prefetch far ahead ($depth = 1$) make the most efficient use of bandwidth. Prefetching improves with the addition of more strides ($1-4$). After this, prefetching slightly farther ahead ($depth = 2$) does slightly increase the number of successful prefetches, but at a high cost in useless prefetches.

VI. Low-locality Applications

In addition to the above studied applications, we also explored the g500-list benchmark from the Graph 500 benchmark suite [53]. This benchmark uses a list data structure to represent the graph, instead of a compressed sparse-row as with g500-csr. The idealized speedup from using all die-stacked memory is 4.4×. Unfortunately, page-caching is entirely ineffective on this application, leading to a 42× *slowdown*. Prefetching improves matters slightly, to “only” a 17× slowdown. The reason g500-list performs so poorly with the page cache is there is almost no locality in the data access stream. In almost all cases, following an edge in the graph involves referencing a new page of data. The work performed per edge-traversal is trivial – a read and update of a small array data-structure. A marginally effective approach we found was static partitioning allocations, which provides 8% speedup.

Rather than include this benchmark throughout the text of our discussion, which would have skewed and obscured the discussion of average benchmark results, we instead have called out its performance separately in this section. While we did not come across another benchmark in our research with such pathologically poor locality, we suspect that many graph algorithms and data structures are like this. For this reason it is critical that any hardware or software designed page cache have mechanisms to bypass the page cache and access off-chip DRAM at the cacheline (or smaller) granularity. As is well known, caches and graph-applications are a poor mix [54].

VII. Conclusion

This paper shows there is a wealth of options for using die-stacked DRAM beyond the hardware-based caches of past work. While a straightforward implementation of a software-based DRAM page cache is ineffective, we showed that by sufficiently reducing the frequency and cost of page migration events (in particular TLB shoot-downs), a software-based DRAM cache can in fact be effective. Varying levels of hardware support (e.g., performing prefetching in hardware vs. software) can improve the effectiveness of the solution, and designers can choose among the range of ideas explored here to best fit the cost and constraints of their target market. This work opens up a wide range of future research to determine the best balance of hardware and software for managing future heterogeneous memory organizations.

Acknowledgments

AMD, the AMD Arrow logo, AMD Opteron, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

[1] J. T. Pawlowski, "Hybrid memory cube," *Hot Chips*, August 2011.

[2] B. Black, "Die stacking is happening," in *International Symposium on Microarchitecture*. ACM, 2013.

[3] G. H. Loh and M. Hill, "Supporting very large DRAM caches with compound-access scheduling and missmap," *Micro, IEEE*, vol. 32, no. 3, pp. 70–78, May 2012.

[4] M. K. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-tags with a simple and practical design," in *International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 235–246.

[5] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked DRAM caches for servers: Hit ratio, latency, or bandwidth? have it all with footprint cache," in *International Symposium on Computer Architecture*. New York, NY, USA: ACM, 2013, pp. 404–415.

[6] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, D. Solihin, and R. Balasubramonian, "CHOP: adaptive filter-based DRAM caching for CMP server platforms," in *International Symposium on High Performance Computer Architecture*, Jan 2010, pp. 1–12.

[7] L. Zhao, R. Iyer, R. Illikkal, and D. Newell, "Exploring dram cache architectures for CMP server platforms," in *25th International Conference on Computer Design*, Oct 2007, pp. 55–62.

[8] HTTP://OSS.SGI.COM/PROJECTS/LIBNUMA/.

[9] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[10] J. Pomerene, T. Puzak, R. Rechtschaffen, and F. Sparacio, "Prefetching system for a cache having a second directory for sequentially accessed blocks," Feb. 21 1989, US Patent 4,807,110.

[11] D. Joseph and D. Grunwald, "Prefetching using markov predictors," *SIGARCH Comput. Archit. News*, vol. 25, no. 2, pp. 252–263, May 1997.

[12] D. H. Kim, K. Athikulwongse, M. Healy, M. Hossain, M. Jung, I. Khorosh, G. Kumar, Y.-J. Lee, D. Lewis, T.-W. Lin, C. Liu, S. Panth, M. Pathak, M. Ren, G. Shen, T. Song, D. H. Woo, X. Zhao, J. Kim, H. Choi, G. Loh, H.-H. Lee, and S.-K. Lim, "3D-MAPS: 3D massively parallel processor with stacked memory," in *International Solid-State Circuits Conference*, Feb 2012, pp. 188–190.

[13] D. Fick, R. Dreslinski, B. Giridhar, G. Kim, S. Seo, M. Fojtik, S. Satpathy, Y. Lee, D. Kim, N. Liu, M. Wieckowski, G. Chen, T. Mudge, D. Sylvester, and D. Blaauw, "Centip3De: A 3930DMIPS/W configurable near-threshold 3d stacked system with 64 ARM Cortex-M3 cores," in *International Solid-State Circuits Conference*, Feb 2012, pp. 190–192.

[14] Y. Deng and W. P. Maly, "Interconnect characteristics of 2.5-D system integration scheme," in *International Symposium on Physical Design*. New York, NY, USA: ACM, 2001, pp. 171–175.

[15] Joint Electron Devices Engineering Council, "JESD235: High Bandwidth Memory (HBM) DRAM," http://www.jedec.org/standards-documents/docs/jesd235.

[16] D. H. Woo, N. H. Seong, D. Lewis, and H.-H. Lee, "An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth," in *International Symposium on High Performance Computer Architecture*, Jan 2010, pp. 1–12.

[17] 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 *International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2012, pp. 247–257.

[18] 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 *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 89–92.

[19] W. Bolosky, R. Fitzgerald, and M. Scott, "Simple but effective techniques for NUMA memory management," *SIGOPS Oper. Syst. Rev.*, vol. 23, no. 5, pp. 19–31, Nov. 1989.

[20] R. P. LaRowe, Jr., C. S. Ellis, and M. A. Holliday, "Evaluation of NUMA memory management through modeling and measurements," *IEEE Trans. Parallel Distrib. Syst.*, vol. 3, no. 6, pp. 686–701, Nov. 1992.

[21] T. Brecht, "On the importance of parallel application placement in NUMA multiprocessors," in *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*. Berkeley, CA, USA: USENIX Association, 1993, pp. 1–1.

[22] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Operating system support for improving data locality on CC-NUMA compute servers," *SIGOPS Oper. Syst. Rev.*, vol. 30, no. 5, pp. 279–289, Sep. 1996.

[23] Z. Majo and T. R. Gross, "Memory system performance in a NUMA multicore multiprocessor," in *International Conference on Systems and Storage*. New York, NY, USA: ACM, 2011, pp. 12:1–12:10.

[24] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A case for NUMA-aware contention management on multicore systems," in *USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2011, pp. 1–1.

[25] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic management: A holistic approach to memory placement on NUMA systems," *SIGARCH Comput. Archit. News*, vol. 41, no. 1, pp. 381–394, Mar. 2013.

[26] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *International Conference on Supercomputing*. New York, NY, USA: ACM, 2011, pp. 85–95.

[27] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Syst.*, vol. 7, no. 4, pp. 321–359, Nov. 1989.

[28] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, “Munin: Distributed shared memory based on type-specific memory coherence,” in *Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 1990, pp. 168–176.

[29] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, “Implementation and performance of Munin,” in *Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 1991, pp. 152–164.

[30] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, “TreadMarks: Distributed shared memory on standard workstations and operating systems,” in *USENIX Winter 1994 Technical Conference*. Berkeley, CA, USA: USENIX Association, 1994, pp. 115–131.

[31] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood, “Fine-grain access control for distributed shared memory,” in *International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 1994, pp. 297–306.

[32] S. K. Reinhardt, J. R. Larus, and D. A. Wood, “Tempest and Typhoon: user-level shared memory,” in *International Symposium on Computer Architecture*, 1994, pp. 325–336.

[33] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. Unsal, “DiDi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory,” in *International Conference on Parallel Architectures and Compilation Techniques*, Oct 2011, pp. 340–349.

[34] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill, “Translation lookaside buffer consistency: A software approach,” in *International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 1989, pp. 113–122.

[35] A. Bhattacharjee and M. Martonosi, “Inter-core cooperative TLB for chip multiprocessors,” in *International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2010, pp. 359–370.

[36] S. Srikantaiah and M. Kandemir, “Synergistic TLBs for high performance address translation in chip multiprocessors,” in *International Symposium on Microarchitecture*, Dec 2010, pp. 313–324.

[37] B. Romanescu, A. Lebeck, D. Sorin, and A. Bracy, “Unified instruction/translation/data (UNITD) coherence: One protocol to rule them all,” in *International Symposium on High Performance Computer Architecture*, Jan 2010, pp. 1–12.

[38] A. Bhattacharjee, D. Lustig, and M. Martonosi, “Shared last-level TLBs for chip multiprocessors,” in *International Symposium on High Performance Computer Architecture*, Feb 2011, pp. 62–63.

[39] ARM, “ARM11 MPCore processor revision 2p0 technical reference,” 2008.

[40] A. Smith, “Sequential program prefetching in memory hierarchies,” *Computer*, vol. 11, no. 12, pp. 7–21, Dec 1978.

[41] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber, “Comparative evaluation of latency reducing and tolerating techniques,” in *International Symposium on Computer Architecture*, 1991, pp. 254–263.

[42] K. Nesbit and J. Smith, “Data cache prefetching using a global history buffer,” in *International Symposium on High Performance Computer Architecture*, Feb 2004.

[43] N. P. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers,” in *International Symposium on Computer Architecture*. New York, NY, USA: ACM, 1990, pp. 364–373.

[44] M. Frigo and S. G. Johnson, “The design and implementation of FFTW3,” *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on “Program Generation, Optimization, and Platform Adaptation”.

[45] G. H. Loh and M. D. Hill, “Efficiently enabling conventional block sizes for very large die-stacked DRAM caches,” in *International Symposium on Microarchitecture*. New York, NY, USA: ACM, 2011, pp. 454–464.

[46] J. S. Liptay, “Structural Aspects of the System/360 Model 85, Part II: The Cache,” *IBM Systems Journal*, vol. 7, no. 1, pp. 15–21, 1968.

[47] B. L. Jacob and T. N. Mudge, “A look at several memory management units, TLB-refill mechanisms, and page table organizations,” in *International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 1998, pp. 295–306.

[48] M.-S. Chang and K. Koh, “Lazy TLB consistency for large-scale multiprocessors,” in *Aizu International Symposium on Parallel Algorithms/Architecture Synthesis*, Mar 1997, pp. 308–315.

[49] Intel, “Multiprocessor specification,” 1997.

[50] ——, “Intel 64 architecture x2 APIC specification,” 2010, HTTP://WWW.INTEL.COM/CONTENT/DAM/DOC/SPECIFICATION-UPDATE/64-ARCHITECTURE-X2APIC-SPECIFICATION.PDF.

[51] D. Lustig, A. Bhattacharjee, and M. Martonosi, “TLB improvements for chip multiprocessors: Inter-core cooperative prefetchers and shared last-level TLBs,” *ACM Trans. Archit. Code Optim.*, vol. 10, no. 1, pp. 2:1–2:38, Apr. 2013.

[52] W. Wong and J.-L. Baer, “The impact of timeliness for hardware-based prefetching from main memory,” 2003, technical Report UW-CSE-02-06-03.

[53] HTTP://WWW.GRAPH500.ORG/.

[54] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, “Latency-tolerant software distributed shared memory,” in *USENIX ATC*, 2015.