Disaggregated Memory for Expansion and Sharing in Blade Servers

Kevin Lim*, Jichuan Chang†, Trevor Mudge*, Parthasarathy Ranganathan†, Steven K. Reinhardt†*, Thomas F. Wenisch*

* University of Michigan, Ann Arbor
{ktlim,tnm,twenisch}@umich.edu

† Hewlett-Packard Labs
{jichuan.chang,partha.ranganathan}@hp.com

+ Advanced Micro Devices, Inc.
steve.reinhardt@amd.com

ABSTRACT

Analysis of technology and application trends reveals a growing imbalance in the peak compute-to-memory-capacity ratio for future servers. At the same time, the fraction contributed by memory systems to total datacenter costs and power consumption during typical usage is increasing. In response to these trends, this paper re-examines traditional compute-memory co-location on a single system and details the design of a new general-purpose architectural building block—a memory blade—that allows memory to be "disaggregated" across a system ensemble. This remote memory blade can be used for memory capacity expansion to improve performance and for sharing memory across servers to reduce provisioning and power costs. We use this memory blade building block to propose two new system architecture solutions—(1) page-swapped remote memory at the virtualization layer, and (2) block-access remote memory with support in the coherence hardware—that enable transparent memory expansion and sharing on commodity-based systems. Using simulations of a mix of enterprise benchmarks supplemented with traces from live datacenters, we demonstrate that memory disaggregation can provide substantial performance benefits (on average 10X) in memory constrained environments, while the sharing enabled by our solutions can improve performance-per-dollar by up to 87% when optimizing memory provisioning across multiple servers.

Categories and Subject Descriptors

C.0 [**Computer System Designs**]: General – *system architectures;* B.3.2 [**Memory Structures**]: Design Styles – *primary memory, virtual memory.*

General Terms

Design, Management, Performance.

Keywords

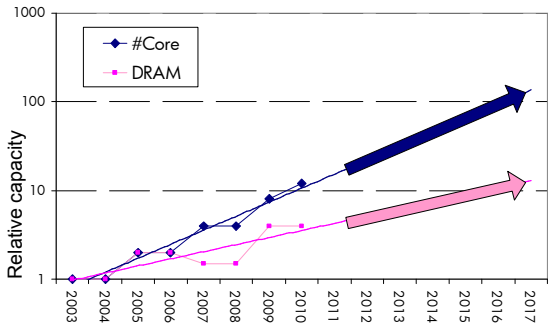
Memory capacity expansion, disaggregated memory, power and cost efficiencies, memory blades.

ISCA'09, June 20–24, 2009, Austin, Texas, USA.

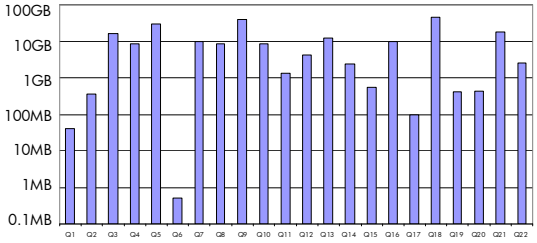
1. INTRODUCTION

Recent trends point to the likely emergence of a new memory wall—one of memory capacity—for future commodity systems. On the *demand* side, current trends point to increased number of cores per socket, with some studies predicting a two-fold increase every two years [1]. Concurrently, we are likely to see an increased number of virtual machines (VMs) per core (VMware quotes 2-4X memory requirements from VM consolidation every generation [2]), and increased memory footprint per VM (e.g., the footprint of Microsoft® Windows® has been growing faster than Moore's Law [3]). However, from a *supply* point of view, the International Technology Roadmap for Semiconductors (ITRS) estimates that the pin count at a socket level is likely to remain constant [4]. As a result, the number of channels per socket is expected to be near-constant. In addition, the rate of growth in DIMM density is starting to wane (2X every three years versus 2X every two years), and the DIMM count per channel is declining (e.g., two DIMMs per channel on DDR3 versus eight for DDR) [5]. Figure 1(a) aggregates these trends to show historical and extrapolated increases in processor computation and associated memory capacity. The processor line shows the projected trend of cores per socket, while the DRAM line shows the projected trend of capacity per socket, given DRAM density growth and DIMM per channel decline. If the trends continue, the growing imbalance between supply and demand may lead to memory capacity per core dropping by 30% every two years, particularly for commodity solutions. If not addressed, future systems are likely to be performance-limited by inadequate memory capacity.

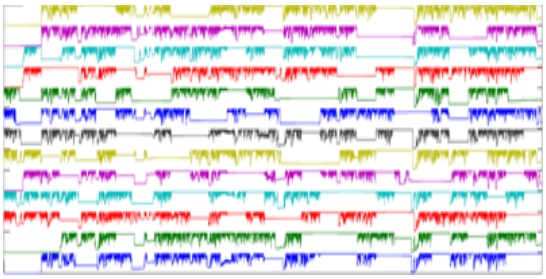
At the same time, several studies show that the contribution of memory to the total costs and power consumption of future systems is trending higher from its current value of about 25% [6][7][8]. Recent trends point to an interesting opportunity to address these challenges—namely that of *optimizing for the ensemble* [9]. For example, several studies have shown that there is significant temporal variation in how resources like CPU time or power are used across applications. We can expect similar trends in memory usage based on variations across application types, workload inputs, data characteristics, and traffic patterns. Figure 1(b) shows how the memory allocated by TPC-H queries can vary dramatically, and Figure 1(c) presents an eye-chart illustration of the time-varying memory usage of 10 randomly-chosen servers from a 1,000-CPU cluster used to render a recent animated feature film [10]. Each line illustrates a server's memory usage varying from a low baseline when idle to the peak memory usage of the application. Rather than provision each system for its worst-case memory usage, a solution that provisions for the



(a) Trends leading toward the memory capacity wall



(b) Memory variations for TPC-H queries (log scale)



(c) Memory variations in server memory utilization

Figure 1: *Motivating the need for memory extension and sharing. (a) On average, memory capacity per processor core is extrapolated to decrease 30% every two years. (b) The amount of granted memory for TPC-H queries can vary by orders of magnitude. (c) "Ensemble" memory usage trends over one month across 10 servers from a cluster used for animation rendering (one of the 3 datacenter traces used in this study).*

typical usage, with the ability to dynamically add memory capacity across the ensemble, can reduce costs and power.

Whereas some prior approaches (discussed in Section 2) can alleviate some of these challenges individually, there is a need for new architectural solutions that can provide *transparent memory capacity expansion to match computational scaling and transparent memory sharing across collections of systems*. In addition, given recent trends towards commodity-based solutions (e.g., [8][9][11]), it is important for these approaches to require at most minor changes to ensure that the low-cost benefits of commodity solutions not be undermined. The increased adoption of blade servers with fast shared interconnection networks and virtualization software creates the opportunity for new memory system designs.

In this paper, we propose a new architectural building block to provide transparent memory expansion and sharing for commodity-based designs. Specifically, we revisit traditional memory designs in which memory modules are co-located with processors on a system board, restricting the configuration and scalability of both compute and memory resources. Instead, we argue for a *disaggregated* memory design that encapsulates an array of commodity memory modules in a separate shared memory blade that can be accessed, as needed, by multiple compute blades via a shared blade interconnect.

We discuss the design of a memory blade and use it to propose two new system architectures to achieve transparent expansion and sharing. Our first solution requires no changes to existing system hardware, using support at the virtualization layer to provide page-level access to a memory blade across the standard PCI Express® (PCIe®) interface. Our second solution proposes minimal hardware support on every compute blade, but provides finer-grained access to a memory blade across a coherent network fabric for commodity software stacks.

We demonstrate the validity of our approach through simulations of a mix of enterprise benchmarks supplemented with traces from three live datacenter installations. Our results show that memory disaggregation can provide significant performance benefits (on average 10X) in memory-constrained environments. Additionally, the sharing enabled by our solutions can enable large improvements in performance-per-dollar (up to 87%) and greater levels of consolidation (3X) when optimizing memory provisioning across multiple servers.

The rest of the paper is organized as follows. Section 2 discusses prior work. Section 3 presents our memory blade design and the implementation of our proposed system architectures, which we evaluate in Section 4. Section 5 discusses other tradeoffs and designs, and Section 6 concludes.

2. RELATED WORK

A large body of prior work (e.g., [12][13][14][15][16][17][18]) has examined using remote servers' memory for swap space [12][16], file system caching [13][15], or RamDisks [14], typically over conventional network interfaces (i.e., Ethernet). These approaches do not fundamentally address the compute-to-memory capacity imbalance: the total memory capacity relative to compute is unchanged when all the servers need maximum capacity at the same time. Additionally, although these approaches can be used to provide sharing, they suffer from significant limitations when targeting commodity-based systems. In particular, these proposals may require substantial system modifications, such as application-specific programming interfaces [18] and protocols [14][17]; changes to the host operating system and device drivers [12][13][14][16]; reduced reliability in the face of remote server crashes [13][16]; and/or impractical access latencies [14][17]. Our solutions target the commodity-based volume server market and thus avoid invasive changes to applications, operating systems, or server architecture.

Symmetric multiprocessors (SMPs) and distributed shared memory systems (DSMs) [19][20][21][22][23][24][25][26][27] allow all the nodes in a system to share a global address space. However, like the network-based sharing approaches, these designs do not target the compute-to-memory-capacity ratio.

Hardware shared-memory systems typically require specialized interconnects and non-commodity components that add costs; in addition, signaling, electrical, and design complexity increase rapidly with system size. Software DSMs [24][25][26][27] can avoid these costs by managing the operations to send, receive, and maintain coherence in software, but come with practical limitations to functionality, generality, software transparency, total costs, and performance [28]. A recent commercial design in this space, Versatile SMP [29], uses a virtualization layer to chain together commodity x86 servers to provide the illusion of a single larger system, but the current design requires specialized motherboards, I/O devices, and non-commodity networking, and there is limited documentation on performance benefits, particularly with respect to software DSMs.

To increase the compute-to-memory ratio directly, researchers have proposed compressing memory contents [30][31] or augmenting/replacing conventional DRAM with alternative devices or interfaces. Recent startups like Virident [32] and Texas Memory [33] propose the use of solid-state storage, such as NAND Flash, to improve memory density albeit with higher access latencies than conventional DRAM. From a technology perspective, fully-buffered DIMMs [34] have the potential to increase memory capacity but with significant trade-offs in power consumption. 3D die-stacking [35] allows DRAM to be placed on-chip as different layers of silicon; in addition to the open architectural issues on how to organize 3D-stacked main memory, this approach further constrains the extensibility of memory capacity. Phase change memory (PCM) is emerging as a promising alternative to increase memory density. However, current PCM devices suffer from several drawbacks that limit their straightforward use as a main memory replacement, including high energy requirements, slow write latencies, and finite endurance. In contrast to our work, none of these approaches enable memory capacity sharing across nodes. In addition, many of these alternatives provide only a one-time improvement, thus delaying but failing to fundamentally address the memory capacity wall.

A recent study [36] demonstrates the viability of a two-level memory organization that can tolerate increased access latency due to compression, heterogeneity, or network access to second-level memory. However, that study does not discuss a commodity implementation for x86 architectures or evaluate sharing across systems. Our prior work [8] employs a variant of this two-level memory organization as part of a broader demonstration of how multiple techniques, including the choice of processors, new packaging design, and use of Flash-based storage, can help improve performance in warehouse computing environments. The present paper follows up on our prior work by: (1) extending the two-level memory design to support x86 commodity servers; (2) presenting two new system architectures for accessing the remote memory; and (3) evaluating the designs on a broad range of workloads and real-world datacenter utilization traces.

As is evident from this discussion, there is currently no single architectural approach that simultaneously addresses memory-to-compute-capacity expansion and memory capacity sharing, and does it in an application/OS-transparent manner on commodity-based hardware and software. The next section describes our approach to define such an architecture.

3. DISAGGREGATED MEMORY

Our approach is based on four observations: (1) The emergence of blade servers with fast shared communication fabrics in the enclosure enables separate blades to share resources across the ensemble. (2) Virtualization provides a level of indirection that can enable OS-and-application-transparent memory capacity changes on demand. (3) Market trends towards commodity-based solutions require special-purpose support to be limited to the non-volume components of the solution. (4) The footprints of enterprise workloads vary across applications and over time; but current approaches to memory system design fail to leverage these variations, resorting instead to worst-case provisioning.

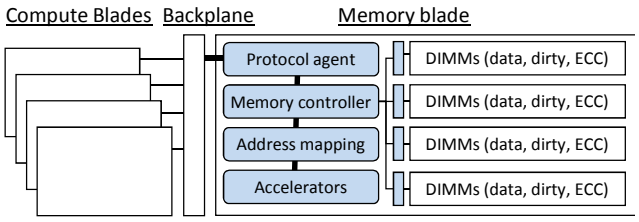
Given these observations, our approach argues for a re-examination of conventional designs that co-locate memory DIMMs in conjunction with computation resources, connected through conventional memory interfaces and controlled through on-chip memory controllers. Instead, we argue for a *disaggregated multi-level design* where we provision an additional separate memory blade, connected at the I/O or communication bus. This memory blade comprises arrays of commodity memory modules assembled to maximize density and cost-effectiveness, and provides extra memory capacity that can be allocated on-demand to individual compute blades. We first detail the design of a memory blade (Section 3.1), and then discuss system architectures that can leverage this component for transparent memory extension and sharing (Section 3.2).

3.1 Memory Blade Architecture

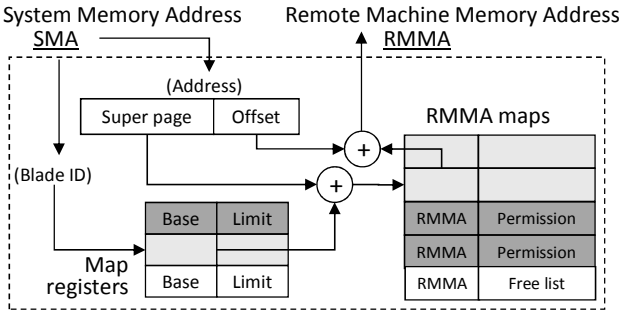
Figure 2(a) illustrates the design of our memory blade. The memory blade comprises a protocol engine to interface with the blade enclosure's I/O backplane interconnect, a custom memory-controller ASIC (or a light-weight CPU), and one or more channels of commodity DIMM modules connected via on-board repeater buffers or alternate fan-out techniques. The memory controller handles requests from client blades to read and write memory, and to manage capacity allocation and address mapping. Optional memory-side accelerators can be added to support functions like compression and encryption.

Although the memory blade itself includes custom hardware, it requires no changes to volume blade-server designs, as it connects through standard I/O interfaces. Its costs are amortized over the entire server ensemble. The memory blade design is straightforward compared to a typical server blade, as it does not have the cooling challenges of a high-performance CPU and does not require local disk, Ethernet capability, or other elements (e.g., management processor, SuperIO, etc.) Client access latency is dominated by the enclosure interconnect, which allows the memory blade's DRAM subsystem to be optimized for power and capacity efficiency rather than latency. For example, the controller can aggressively place DRAM pages into active power-down mode, and can map consecutive cache blocks into a single memory bank to minimize the number of active devices at the expense of reduced single-client bandwidth. A memory blade can also serve as a vehicle for integrating alternative memory technologies, such as Flash or phase-change memory, possibly in a heterogeneous combination with DRAM, without requiring modification to the compute blades.

To provide protection and isolation among shared clients, the memory controller translates each memory address accessed by a



(a) Memory blade design



(b) Address mapping

Figure 2: Design of the memory blade. *(a) The memory blade connects to the compute blades via the enclosure backplane. (b) The data structures that support memory access and allocation/revocation operations.*

client blade into an address local to the memory blade, called the Remote Machine Memory Address (RMMA). In our design, each client manages both local and remote physical memory within a single System Memory Address (SMA) space. Local physical memory resides at the bottom of this space, with remote memory mapped at higher addresses. For example, if a blade has 2 GB of local DRAM and has been assigned 6 GB of remote capacity, its total SMA space extends from 0 to 8 GB. Each blade's remote SMA space is mapped to a disjoint portion of the RMMA space. This process is illustrated in Figure 2(b). We manage the blade's memory in large chunks (e.g., 16 MB) so that the entire mapping table can be kept in SRAM on the memory blade's controller. For example, a 512 GB memory blade managed in 16 MB chunks requires only a 32K-entry mapping table. Using these "superpage" mappings avoids complex, high-latency DRAM page table data structures and custom TLB hardware. Note that providing shared-memory communications among client blades (as in distributed shared memory) is beyond the scope of this paper.

Allocation and revocation: The memory blade's total capacity is partitioned among the connected clients through the cooperation of the virtual machine monitors (VMMs) running on the clients, in conjunction with enclosure-, rack-, or datacenter-level management software. The VMMs in turn are responsible for allocating remote memory among the virtual machine(s) (VMs) running on each client system. The selection of capacity allocation policies, both among blades in an enclosure and among VMs on a blade, is a broad topic that deserves separate study. Here we restrict our discussion to designing the mechanisms for allocation and revocation.

Allocation is straightforward: privileged management software on the memory blade assigns one or more unused memory blade superpages to a client, and sets up a mapping from the chosen blade ID and SMA range to the appropriate RMMA range.

In the case where there are no unused superpages, some existing mapping must be revoked so that memory can be reallocated. We assume that capacity reallocation is a rare event compared to the frequency of accessing memory using reads and writes. Consequently, our design focuses primarily on correctness and transparency and not performance.

When a client is allocated memory on a fully subscribed memory blade, management software first decides which other clients must give up capacity, then notifies the VMMs on those clients of the amount of remote memory they must release. We propose two general approaches for freeing pages. First, most VMMs already provide paging support to allow a set of VMs to oversubscribe local memory. This paging mechanism can be invoked to evict local or remote pages. When a remote page is to be swapped out, it is first transferred temporarily to an empty local frame and then paged to disk. The remote page freed by this transfer is released for reassignment.

Alternatively, many VMMs provide a "balloon driver" [37] within the guest OS to allocate and pin memory pages, which are then returned to the VMM. The balloon driver increases memory pressure within the guest OS, forcing it to select pages for eviction. This approach generally provides better results than the VMM's paging mechanisms, as the guest OS can make a more informed decision about which pages to swap out and may simply discard clean pages without writing them to disk. Because the newly freed physical pages can be dispersed across both the local and remote SMA ranges, the VMM may need to relocate pages within the SMA space to free a contiguous 16 MB remote superpage.

Once the VMMs have released their remote pages, the memory blade mapping tables may be updated to reflect the new allocation. We assume that the VMMs can generally be trusted to release memory on request; the unlikely failure of a VMM to release memory promptly indicates a serious error and can be resolved by rebooting the client blade.

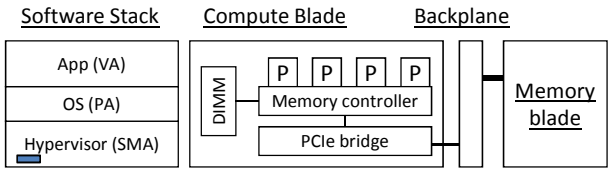
3.2 System Architecture with Memory Blades

Whereas our memory-blade design enables several alternative system architectures, we discuss two specific designs, one based on page swapping and another using fine-grained remote access. In addition to providing more detailed examples, these designs also illustrate some of the tradeoffs in the multi-dimensional design space for memory blades. Most importantly, they compare the method and granularity of access to the remote blade (page-based versus block-based) and the interconnect fabric used for communication (PCI Express versus HyperTransport).

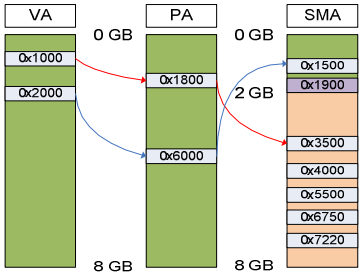
3.2.1 Page-Swapping Remote Memory (PS)

Our first design avoids any hardware changes to the high-volume compute blades or enclosure; the memory blade itself is the only non-standard component. This constraint implies a conventional I/O backplane interconnect, typically PCIe. This basic design is illustrated in Figure 3(a).

Because CPUs in a conventional system cannot access cacheable memory across a PCIe connection, the system must bring locations into the client blade's local physical memory before they



(a) Compute blade



(b) Address mapping process

Figure 3: *Page-swapping remote memory system design. (a) No changes are required to compute servers and networking on existing blade designs. Our solution adds minor modules (shaded block) to the virtualization layer. (b) The address mapping design places the extended capacity at the top of the address space.*

can be accessed. We leverage standard virtual-memory mechanisms to detect accesses to remote memory and relocate the targeted locations to local memory on a page granularity. In addition to enabling the use of virtual memory support, page-based transfers exploit locality in the client's access stream and amortize the overhead of PCIe memory transfers.

To avoid modifications to application and OS software, we implement this page management in the VMM. The VMM detects accesses to remote data pages and swaps those data pages to local memory before allowing a load or store to proceed.

Figure 3(b) illustrates our page management scheme. Recall that, when remote memory capacity is assigned to a specific blade, we extend the SMA (machine physical address) space at that blade to provide local addresses for the additional memory. The VMM assigns pages from this additional address space to guest VMs, where they will in turn be assigned to the guest OS or to applications. However, because these pages cannot be accessed directly by the CPU, the VMM cannot set up valid page-table entries for these addresses. It instead tracks the pages by using "poisoned" page table entries without their valid bits set or by tracking the mappings outside of the page tables (similar techniques have been used to prototype hybrid memory in VMWare [38]). In either case, a direct CPU access to remote memory will cause a page fault and trap into the VMM. On such a trap, the VMM initiates a page swap operation. This simple OS-transparent memory-to-memory page swap should not be confused with OS-based virtual memory swapping (paging to swap space), which is orders of magnitude slower and involves an entirely different set of sophisticated data structures and algorithms.

In our design, we assume page swapping is performed on a 4 KB granularity, a common page size used by operating systems. Page swaps logically appear to the VMM as a swap from high SMA addresses (beyond the end of local memory) to low addresses (within local memory). To decouple the swap of a remote page to local memory and eviction of a local page to remote memory, we maintain a pool of free local pages for incoming swaps. The software fault handler thus allocates a page from the local free list and initiates a DMA transfer over the PCIe channel from the remote memory blade. The transfer is performed synchronously (i.e., the execution thread is stalled during the transfer, but other threads may execute). Once the transfer is complete, the fault handler updates the page table entry to point to the new, local SMA address and puts the prior remote SMA address into a pool of remote addresses that are currently unused.

To maintain an adequate supply of free local pages, the VMM must occasionally evict local pages to remote memory, effectively performing the second half of the logical swap operation. The VMM selects a high SMA address from the remote page free list and initiates a DMA transfer from a local page to the remote memory blade. When complete, the local page is unmapped and placed on the local free list. Eviction operations are performed asynchronously, and do not stall the CPU unless a conflicting access to the in-flight page occurs during eviction.

3.2.2 *Fine-Grained Remote Memory Access* (FGRA)

The previous solution avoids any hardware changes to the commodity compute blade, but at the expense of trapping to the VMM and transferring full pages on every remote memory access. In our second approach, we examine the effect of a few minimal hardware changes to the high-volume compute blade to enable an alternate design that has higher performance potential. In particular, this design allows CPUs on the compute blade to access remote memory directly at cache-block granularity.

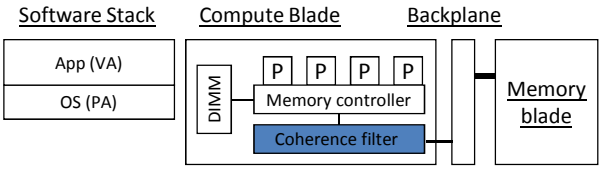


Figure 4: *Fine-grained remote memory access system design. This design assumes minor coherence hardware support in every compute blade.*

Our approach leverages the glueless SMP support found in current processors. For example, AMD Opteron™ processors have up to three coherent HyperTransport™ links coming out of the socket. Our design, shown in Figure 4, uses custom hardware on the compute blade to redirect cache fill requests to the remote memory blade. Although it does require custom hardware, the changes to enable our FGRA design are relatively straightforward adaptations of current coherent memory controller designs

This hardware, labeled "Coherence filter" in Figure 4, serves two purposes. First, it selectively forwards only necessary coherence protocol requests to the remote memory blade. For example, because the remote blade does not contain any caches, the coherence filter can respond immediately to invalidation requests. Only memory read and write requests require processing at the remote memory blade. In the terminology of glueless x86

multiprocessors, the filter ensures that the memory blade is a home agent but not a cache agent. Second, the filter can optionally translate coherence messages destined for the memory blade into an alternate format. For example, HyperTransport-protocol read and write requests can be translated into generic PCIe commands, allowing the use of commodity backplanes and decoupling the memory blade from specific cache-coherence protocols and processor technologies.

Because this design allows the remote SMA space to be accessed directly by CPUs, VMM support is not required; an unmodified OS can treat both local and remote addresses uniformly. However, a VMM or additional OS support is required to enable dynamic allocation or revocation of remote memory. Performance can also potentially be improved by migrating the most frequently accessed remote pages into local memory, swapping them with infrequently accessed local pages—a task that could be performed by a VMM or by extending the NUMA support available in many OSes.

4. EVALUATION

4.1 Methodology

We compare the performance of our memory-blade designs to a conventional system primarily via trace-based simulation. Using traces rather than a detailed execution-driven CPU model makes it practical to process the billions of main-memory references needed to exercise a multi-gigabyte memory system. Although we forgo the ability to model overlap between processor execution and remote memory accesses with our trace-based simulations, our memory reference traces are collected from a simulator that does model overlap of local memory accesses. Additionally, we expect overlap for remote accesses to be negligible due to the relatively high latencies to our remote memory blade.

We collected memory reference traces from a detailed full-system simulator, used and validated in prior studies [39], modified to record the physical address, CPU ID, timestamp and read/write status of all main-memory accesses. To make it feasible to run the workloads to completion, we use a lightweight CPU model for this simulation. (Each simulation still took 1-2 weeks to complete.) The simulated system has four 2.2 GHz cores, with per-core dedicated 64KB L1 and 512 KB L2 caches, and a 2 MB L3 shared cache.

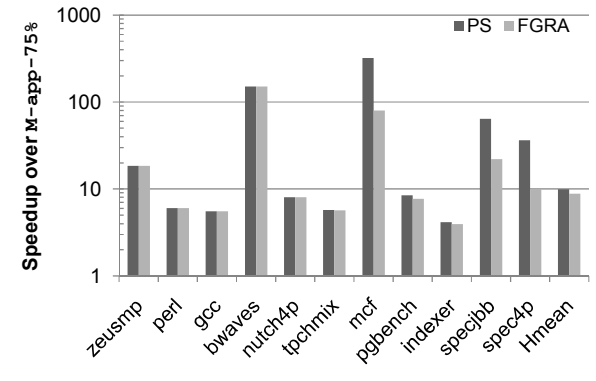
The common simulation parameters for our remote memory blade are listed in Table 1. For the baseline `PS`, we assume that the memory blade interconnect has a latency of 120 ns and bandwidth of 1 GB/s (each direction), based loosely on a PCIe 2.0 x2 channel. For the baseline `FGRA`, we assume a more aggressive channel, e.g., based on HyperTransport™ or a similar technology, with 60 ns latency and 4 GB/s bandwidth. Additionally, for `PS`, each access to remote memory results in a trap to the VMM, and VMM software must initiate the page transfer. Based on prior work [40], we assume a total of 330 ns (roughly 1,000 cycles on a 3 GHz processor) for this software overhead, including the trap itself, updating page tables, TLB shootdown, and generating the request message to the memory blade. All of our simulated systems are modeled with a hard drive with 8 ms access latency and 50 MB/s sustained bandwidth. We perform initial data placement using a first-touch allocation policy.

We validated our model on a real machine to measure the impact of reducing the physical memory allocation in a conventional server. We use an HP c-Class BL465c server with 2.2GHz AMD Opteron 2354 processors and 8 GB of DDR2-800 DRAM. To model a system with less DRAM capacity, we force the Linux kernel to reduce physical memory capacity using a boot-time kernel parameter.

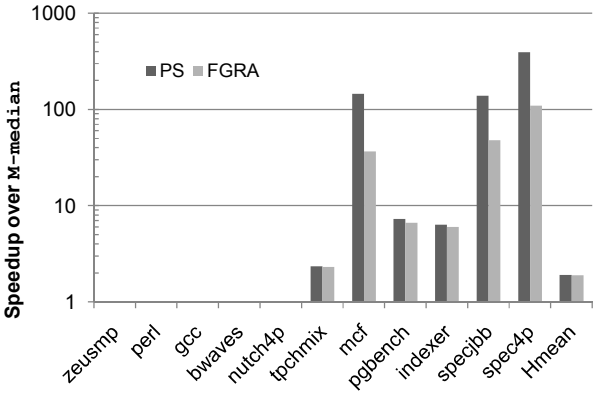
The workloads used to evaluate our designs include a mixture of Web 2.0-based benchmarks (*nutch, indexer*), traditional server benchmarks (*pgbench, TPC-H, SPECjbb®2005*), and traditional computational benchmarks (*SPEC® CPU2006 – zeusmp, gcc, perl, bwaves, mcf*). Additionally we developed a multi-programmed workload, *spec4p*, by combining the traces from *zeusmp, gcc, perl,* and *mcf*. *Spec4p* offers insight into multiple workloads sharing a single server's link to the memory blade. Table 1 describes these workloads in more detail. We further broadly classify the workloads into three groups—low, medium, and high—based on their memory footprint sizes. The low group consists of benchmarks whose footprint is less than 1 GB,

Memory blade parameters					
DRAM Latency	120 ns	Map table access	5 ns	Request packet processing	60 ns
DRAM Bandwidth	6.4 GB/s	Transfer page size	4KB	Response packet processing	60 ns
Workloads					**Footprint size**
SPEC CPU 2006	5 large memory benchmarks: *zeusmp*, *perl*, *gcc*, *bwaves*, and *mcf*, as well as a combination of four of them, *spec4p*.				Low (zeusmp, gcc, perl, bwaves), Medium (mcf), High (spec4p)
nutch4p	Nutch 0.9.1 search engine with Resin and Sun JDK 1.6.0, 5GB index hosted on tempfs.				Medium
tpchmix	TPC-H running on MySQL 5.0 with scaling factor of 1. 2 copies of query 17 mixed with query 1 and query 3 (representing balanced, scan and join heavy queries).				Medium
pgbench	TPC-B like benchmark running PostgreSQL 8.3 with pgbench and a scaling factor of 100.				High
Indexer	Nutch 0.9.1 indexer, Sun JDK 1.6.0 and HDFS hosted on one hard drive.				High
SPECjbb	4 copies of Specjbb 2005, each with 16 warehouses, using Sun JDK 1.6.0.				High
Real-world traces					
Animation	Resource utilization traces collected on 500+ animation rendering servers over a year, 1-second sample interval. We present data from traces from a group of 16 representative servers.				
VM consolidation	VM consolidation traces of 16 servers based on enterprise and web2.0 workloads, maximum resource usage reported every 10-minute interval.				
Web2.0	Resource utilization traced collected on 290 servers from a web2.0 company, we use sar traces with 1-second sample interval for 16 representative servers.				

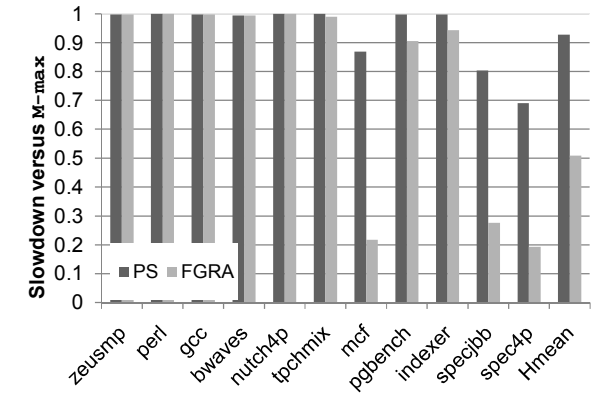
Table 1: *Simulation parameters and workload/trace description.*



(a) Speedup over `M-app-75%` provisioning



(b) Speedup over `M-median` provisioning



(c) Slowdown vs. worst-case (`M-max`) provisioning

Figure 5: Capacity expansion results. *(a) and (b) show the performance improvement for our two designs over memory-capacity-constrained baselines; (c) shows performance and costs relative to worst-case provisioning.*

medium ranges from 1 GB to 1.75 GB, and high includes those with footprints between 1.75GB and 3GB. In addition to these workloads, we have also collected traces of memory usage in three real-world, large-scale datacenter environments. These environments include *Animation*, *VM consolidation*, and *web2.0*, and are described in Table 1. These traces were each gathered for over a month across a large number of servers and are used to guide our selection of workloads to mimic the time-varying memory requirements of applications seen in real-world environments.

To quantify the cost benefits of our design, we developed a cost model for our disaggregated memory solutions and the baseline servers against which we compare. Because our designs target the memory system, we present data specific to the memory system. We gathered price data from public and industry sources for as many components as possible. For components not available off the shelf, such as our remote memory blade controller, we estimate a cost range. We further include power and cooling costs, given a typical 3-year server lifespan. We used DRAM power calculators to evaluate the power consumption of DDR2 devices. Estimates for the memory contributions towards power and cooling are calculated using the same methodology as in [8].

4.2 Results

4.2.1 Memory expansion for individual benchmarks

We first focus on the applicability of memory disaggregation to address the memory capacity wall for individual benchmarks. To illustrate scenarios where applications run into memory capacity limitations due to a core-to-memory ratio imbalance, we perform an experiment where we run each of our benchmarks on a baseline system with only 75% of that benchmark's memory footprint (`M-app-75%`). The baseline system must swap pages to disk to accommodate the full footprint of the workload. We compare these with our two disaggregated-memory architectures, `PS` and `FGRA`. In these cases, the compute nodes continue to have local DRAM capacity corresponding to only 75% of the benchmark's memory footprint, but have the ability to exploit capacity from a remote memory blade. We assume 32GB of memory on the memory blade, which is sufficient to fit any application's footprint. Figure 5(a) summarizes the speedup for the `PS` and `FGRA` designs relative to the baseline. Both of our new solutions achieve significant improvements, ranging from 4X to 320X higher performance. These improvements stem from the much lower latency of our remote memory solutions compared to OS-based disk paging. In particular, *zeusmp, bwaves, mcf, specjbb*, and *spec4p* show the highest benefits due to their large working sets.

Interestingly, we also observe that `PS` outperforms `FGRA` in this experiment, despite our expectations for `FGRA` to achieve better performance due to its lower access latency. Further investigation reveals that the page swapping policy in `PS`, which transfers pages from remote memory to local memory upon access, accounts for its performance advantage. Under PS, although the initial access to a remote memory location incurs a high latency due to the VMM trap and the 4 KB page transfer over the slower PCIe interconnect, subsequent accesses to that address consequently incur only local-memory latencies. The `FGRA` design, though it has lower remote latencies compared to `PS`, continues to incur these latencies for every access to a frequently used remote location. Nevertheless, `FGRA` still outperforms the baseline. We examine the addition of page swapping to `FGRA` in Section 4.2.5.

Figure 5(b) considers an alternate baseline where the compute server memory is set to approximate the median-case memory footprint requirements across our benchmarks (`M-median` = 1.5GB). This baseline models a realistic scenario where the server

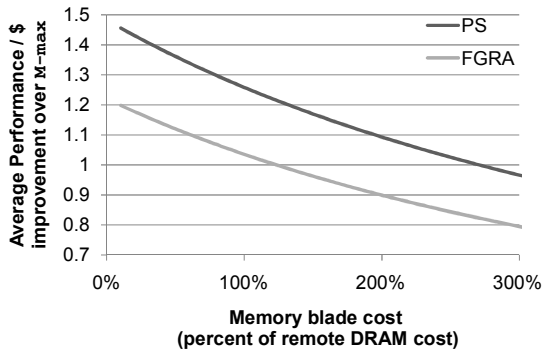


Figure 6: Memory blade cost analysis. *Average performance-per-memory dollar improvement versus memory blade costs relative to the total cost of remote DRAM.*

is provisioned for the common-case workload, but can still see a mix of different workloads. Figure 5(b) shows that our proposed solutions now achieve performance improvements only for benchmarks with high memory footprints. For other benchmarks, the remote memory blade is unused, and does not provide any benefit. More importantly, it does not cause any slowdown.

Finally, Figure 5(c) considers a baseline where the server memory is provisioned for the worst-case application footprint (`M-max` = 4GB). This baseline models many current datacenter scenarios where servers are provisioned in anticipation of the worst-case load, either across workloads or across time. We configure our memory disaggregation solutions as in the previous experiment, with `M-median` provisioned per-blade and additional capacity in the remote blade. Our results show that, for workloads with small footprints, our new solutions perform comparably. For workloads with larger footprints, going to remote memory causes a slowdown compared to local memory; however, `PS` provides comparable performance in some large-footprint workloads (*pgbench*, *indexer*), and on the remaining workloads its performance is still within 30% of `M-max`. As before, `FGRA` loses performance as it does not exploit locality patterns to ensure most accesses go to local memory.

4.2.2 Power and cost analysis

Using the methodology described in 4.1, we estimate the memory power draw of our baseline `M-median` system as 10 W, and our `M-max` system as 21 W. To determine the power draw of our disaggregated memory solutions, we assume local memory provisioned for median capacity requirements (as in `M-median`) and a memory blade with 32 GB shared by 16 servers. Furthermore, because the memory blade can tolerate increased DRAM access latency, we assume it aggressively employs DRAM low-power sleep modes. For a 16-server ensemble, we estimate the amortized per-server memory power of the disaggregated solution (including all local and remote memory and the memory blade interface hardware, such as its controller, and I/O connections) at 15 W.

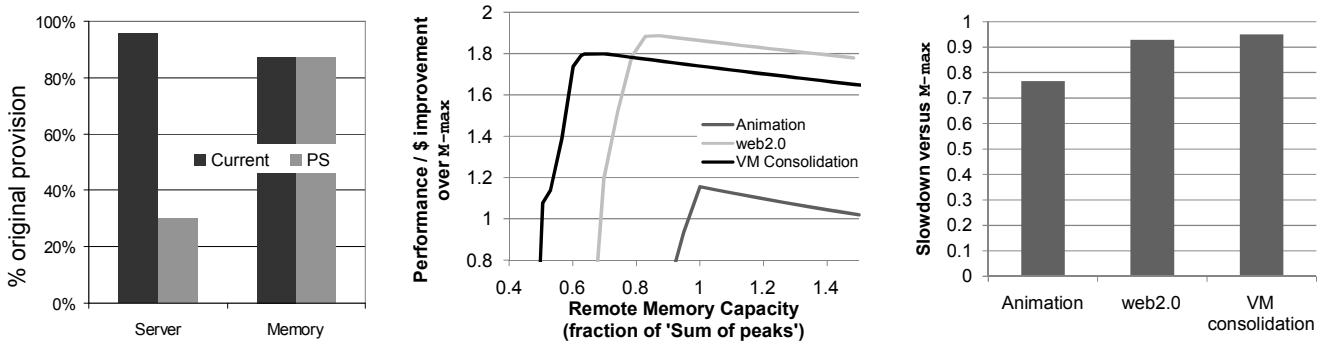
Figure 6 illustrates the cost impact of the custom designed memory blade, showing the changes in the average performance-per-memory cost improvement over the baseline `M-max` system as memory blade cost varies. To put the memory blade cost into context with the memory subsystem, the cost is calculated as a percentage of the total remote DRAM costs (memory blade cost divided by remote DRAM costs), using 32 GB of remote memory. Note that for clarity, the cost range on the horizontal axis refers only to the memory blade interface/packaging hardware excluding DRAM costs (the fixed DRAM costs are factored in to the results). The hardware cost break-even points for `PS` and `FGRA` are high, implying a sufficiently large budget envelope for the memory blade implementation. We expect that the overhead of a realistic implementation of a memory blade could be below 50% of the remote DRAM cost (given current market prices). This overhead can be reduced further by considering higher capacity memory blades; for example, we expect the cost to be below 7% of the remote DRAM cost of a 256 GB memory blade.

4.2.3 Server consolidation

Viewed as a key application for multi-core processors, server consolidation improves hardware resource utilization by hosting multiple virtual machines on a single physical platform. However, memory capacity is often the bottleneck to server consolidation because other resources (e.g., processor and I/O) are easier to multiplex, and the growing imbalance between processor and memory capacities exacerbates the problem. This effect is evident in our real-world *web2.0* traces, where processor utilization rates are typically below 30% (rarely over 45%) while more than 80% of memory is allocated, indicating limited consolidation opportunities without memory expansion. To address this issue, current solutions either advocate larger SMP servers for their memory capacity or sophisticated hypervisor memory management policies to reduce workload footprints, but they incur performance penalties, increase costs and complexity, and do not address the fundamental processor-memory imbalance.

Memory disaggregation enables new consolidation opportunities by supporting processor-independent memory expansion. With memory blades to provide the second-level memory capacity, we can reduce each workload's processor-local memory allocation to less than its total footprint (`M-max`) while still maintaining comparable performance (i.e., <3% slowdown). This workload-specific local vs. remote memory ratio determines how much memory can be freed on a compute server (and shifted onto the memory blade) to allow further consolidation. Unfortunately, it is not possible to experiment in production datacenters to determine these ratios. Instead, we determine the typical range of local-to-remote ratios using our simulated workload suite. We can then use this range to investigate the potential for increased consolidation using resource utilization traces from production systems.

We evaluate the consolidation benefit using the *web2.0* workload (CPU, memory and IO resource utilization traces for 200+ servers) and a sophisticated consolidation algorithm similar to that used by Rolia et al. [41]. The algorithm performs multi-dimensional bin packing to minimize the number of servers needed for given resource requirements. We do not consider the other two traces for this experiment. *Animation* is CPU-bound and runs out of CPU before it runs out of memory, so memory disaggregation does not help. However, as CPU capacity increases in the future, we may likely encounter a similar situation as *web2.0*. *VM consolidation*, on the other hand, does run out of memory before it runs of out CPU, but these traces already represent the result of consolidation, and in the absence of information on the prior consolidation policy, it is hard to make a fair determination of the baseline and the additional benefits from memory disaggregation over existing approaches.



(a) Reductions from consolidation (b) Cost efficiency vs. remote capacity (c) Perf. at cost-optimized provisioning

Figure 7: Mixed workload and ensemble results. *(a) Hardware reductions from improved VM consolidation made possible by remote memory. (b) Performance-per-dollar as remote memory capacity is varied. (c) Slowdown relative to per-blade worst-case provisioning (`M-max`) at cost-optimal provisioning.*

As shown in Figure 7(a), without memory disaggregation, the state-of-the-art algorithm ("Current") achieves only modest hardware reductions (5% processor and 13% memory); limited memory capacity precludes further consolidation. In contrast, page-swapping–based memory disaggregation corrects the time-varying imbalance between VM memory demands and local capacity, allowing a substantial reduction of processor count by a further 68%.

4.2.4 Ensemble-level memory sharing

We now examine the benefits of disaggregated memory in multi-workload server ensembles with time-varying requirements. By dynamically sharing memory capacity at an ensemble level, disaggregated memory can potentially exploit the inter- and intra-workload variations in memory requirements. This variation is highlighted by the difference in the *peak of sums* versus the *sum of peaks.* The *peak of sums* is the maximum total memory required across the ensemble at any single point in time. On the other hand, the *sum of peaks* is the sum of the worst-case memory requirements of all the servers for the applications they are running. In conventional environments, servers must be provisioned for the worst-case memory usage (sum of peaks) to avoid potentially-catastrophic performance losses from underprovisioning (which may lead to swapping/thrashing). However, the peak of sums is often much smaller than the sum of peaks as servers rarely reach their peak loads simultaneously; systems provisioned for worst-case demands are nearly always underutilized. Ensemble-level sharing allows servers to instead be provisioned for the sum of peaks, saving costs and power.

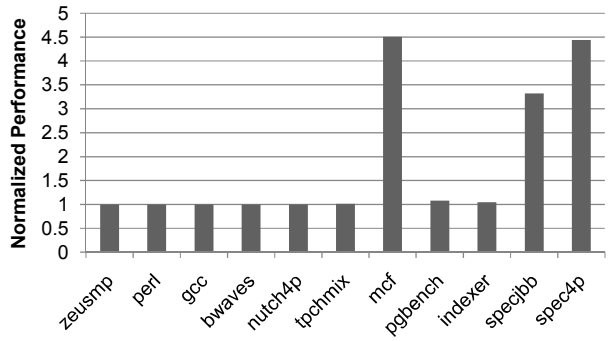
We examine the potential of ensemble-level sharing for a 16-server blade enclosure running a mix of enterprise workloads with varying memory requirements (similar to the scenario shown in Figure 1(b)). We examine three real-world enterprise datacenter workload traces (*Animation, VM consolidation*, and *web2.0*), and create a mixed workload trace using our simulated workloads to mimic the same memory usage patterns. We divide each trace into epochs and measure the *processing done per epoch*; we then compare these rates across different configurations to estimate performance benefits. Given that allocation policies are outside the scope of this paper, we assume a simple policy where, at the beginning of each epoch, each compute blade requests the additional memory it needs from the memory blade. (In a task-scheduling environment, this could be based on prior knowledge of the memory footprint of the new task that will be scheduled.) For the cost of the memory blade, we conservatively estimated the price to be approximately that of a low-end system. We expect this estimate to be conservative because of the limited functionality and hardware requirements of the memory blade versus that of a general purpose server.

Figure 7(b) shows the performance-per-memory-dollar improvement, normalized to the `M-max` baseline, for `PS` over a range of remote memory sizes. We focus on the `PS` design as the `FGRA` design is not as competitive due to its inability to migrate frequently accessed data to local memory (see Section 4.2.1). As is shown, both the *VM consolidation* and *web2.0* traces benefit substantially from ensemble-level provisioning, gaining 78% and 87% improvement in performance-per-dollar while requiring only 70% and 85% of the sum-of-peaks memory capacity, respectively. These savings indicate that the remote memory capacity can be reduced below worst-case provisioning (sum of peaks) because demands in these workloads rarely reach their peak simultaneously. In contrast, the peak of sums closely tracks the sum of peaks in the *Animation* trace, limiting the opportunity for cost optimization.

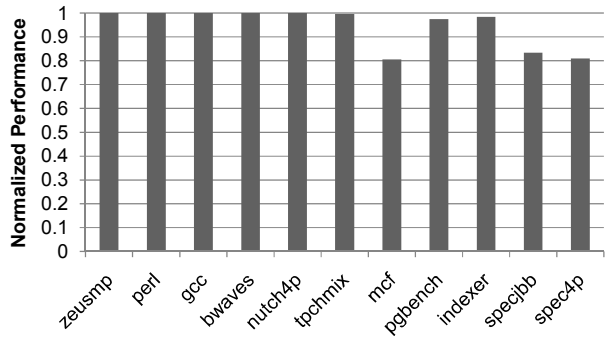
We next evaluate the performance of a cost-optimized disaggregated memory solution relative to the `M-max` baseline (worst-case provisioning). Figure 7(c) shows the performance sacrificed by the per-workload cost-optimal design (as determined by the performance-per-dollar peak for each workload in Figure 7(b)). There is minimal performance loss for the *web2.0* and *VM consolidation* traces (5% and 8%), indicating that disaggregated memory can significantly improve cost-efficiency without adversely affecting performance. For the *Animation* traces there is a larger performance penalty (24%) due to its consistently high memory demands. Compared to the `M-median` baseline, the disaggregated memory designs show substantial throughput improvements (34-277X) for all the traces.

4.2.5 Alternate designs

As discussed earlier, our `FGRA` design suffers relative to `PS` because it does not exploit locality by swapping heavily used remote pages to local memory. This disadvantage can be



(a) FGRA placement-aware design



(b) FGRA over PCIe design

Figure 8: Alternate FGRA designs. (a) shows the normalized performance when FGRA is supplemented by NUMA-type optimizations; (b) shows the performance loss from tunneling FGRA accesses over a commodity interconnect.

addressed by adding page migration to FGRA, similar to existing CC-NUMA optimizations (e.g., Linux's memory placement optimizations [42]). To study the potential impact of this enhancement, we modeled a hypothetical system that tracks page usage and, at 10 ms intervals, swaps the most highly used pages into local memory. Figure 8(a) summarizes the speedup of this system over the base FGRA design for M-median compute blades. For the high-footprint workloads that exhibit the worst performance with FGRA (*mcf, SPECjbb, and SPEC4p*), page migration achieves 3.3-4.5X improvement, with smaller (5-8%) benefit on other high-footprint workloads. For all workloads, the optimized FGRA performs similarly to, and in a few cases better than, PS. These results motivate further examination of data placement policies for FGRA.

The hardware cost of FGRA can be reduced by using a standard PCIe backplane (as PS does) rather than a coherent interconnect, as discussed in Section 3.2.2. This change incurs a latency and bandwidth penalty as the standardized PCIe interconnect is less aggressive than a more specialized interconnect such as cHT. Figure 8(b) shows the change in performance relative to the baseline FGRA. Performance is comparable, decreasing by at most 20% on the higher memory usage workloads. This performance loss may be acceptable if the cost extending a high-performance interconnect like cHT across the enclosure backplane is high.

Though not shown here (due to space constraints), we have also studied sensitivity of our results to the VMM overhead and memory latency parameters in Table 1. Our results show no qualitative change to our conclusions.

5. DISCUSSION

Evaluation assumptions. Our evaluation does not model interconnect routing, arbitration, buffering, and QoS management in detail. Provided interconnect utilization is not near saturation, these omissions will not significantly impact transfer latencies. We have confirmed that per-blade interconnect bandwidth consumption falls well below the capabilities of PCIe and HT. However, the number of channels to the memory blade may need to be scaled with the number of supported clients.

Impact of the memory blade on ensemble manageability. Memory disaggregation has both positive and negative impacts on enterprise system reliability, availability, security, and manageability. From a reliability perspective, dynamic reprovisioning provides an inexpensive means to equip servers with hot-spare DRAM; in the event of a DIMM failure anywhere in the ensemble, memory can be remapped and capacity reassigned to replace the lost DIMM. However, the memory blade also introduces additional failure modes that impact multiple servers. A complete memory-blade failure might impact several blades, but this possibility can be mitigated by adding redundancy to the blade's memory controller. We expect that high availability could be achieved at a relatively low cost, given the controller's limited functionality. To provide security and isolation, our design enforces strict assignment of capacity to specific blades, prohibits sharing, and can optionally erase memory content prior to reallocation to ensure confidentiality. From a manageability perspective, disaggregation allows management software to provision memory capacity across blades, reducing the need to physically relocate DIMMs.

Memory blade scalability and sharing. There are several obvious extensions to our designs. First, to provide memory scaling beyond the limits of a single memory blade, a server ensemble might include multiple memory blades. Second, prior studies of consolidated VMs have shown substantial opportunities to reduce memory requirements via copy-on-write content-based page sharing across VMs [37]. Disaggregated memory offers an even larger scope for sharing content across multiple compute blades. Finally, in some system architectures, subsets of processors/blades share a memory coherence domain, which we might seek to extend via disaggregation.

Synergy with emerging technologies. Disaggregated memory extends the conventional virtual memory hierarchy with a new layer. This layer introduces several possibilities to integrate new technologies into the ensemble memory system that might prove latency- or cost-prohibitive in conventional blade architectures. First, we foresee substantial opportunity to leverage emerging interconnect technologies (e.g., optical interconnects) to improve communication latency and bandwidth and allow greater physical distance between compute and memory blades. Second, the memory blade's controller provides a logical point in the system hierarchy to integrate accelerators for capacity and reliability enhancements, such as memory compression [30][31]. Finally, one might replace or complement memory blade DRAM with higher-density, lower-power, and/or non-volatile memory

technologies, such as NAND Flash or phase change memory. Unlike conventional memory systems, where it is difficult to integrate these technologies because of large or asymmetric access latencies and lifetime/wearout challenges, disaggregated memory is more tolerant of increased access latency, and the memory blade controller might be extended to implement wear-leveling and other lifetime management strategies [43]. Furthermore, disaggregated memory offers the potential for transparent integration. Because of the memory interface abstraction provided by our design, Flash or phase change memory can be utilized on the memory blade without requiring any further changes on the compute blade.

6. CONCLUSIONS

Constraints on per-socket memory capacity and the growing contribution of memory to total datacenter costs and power consumption motivate redesign of the memory subsystem. In this paper, we discuss a new architectural approach—memory disaggregation—which uses dedicated memory blades to provide OS-transparent memory extension and ensemble sharing for commodity-based blade-server designs. We propose an extensible design for the memory blade, including address remapping facilities to support protected dynamic memory provisioning across multiple clients, and unique density optimizations to address the compute-to-memory capacity imbalance. We discuss two different system architectures that incorporate this blade: a page-based design that allows memory blades to be used on current commodity blade server architectures with small changes to the virtualization layer, and an alternative that requires small amounts of extra hardware support in current compute blades but supports fine-grained remote accesses and requires no changes to the software layer. To the best of our knowledge, our work is the first to propose a commodity-based design that simultaneously addresses compute-to-memory capacity extension and cross-node memory capacity sharing. We are also the first to consider dynamic memory sharing across the I/O communication network in a blade enclosure and quantitatively evaluate design tradeoffs in this environment.

Simulations based on detailed traces from 12 enterprise benchmarks and three real-world enterprise datacenter deployments show that our approach has significant potential. The ability to extend and share memory can achieve orders of magnitude performance improvements in cases where applications run out of memory capacity, and similar orders of magnitude improvement in performance-per-dollar in cases where systems are overprovisioned for peak memory usage. We also demonstrate how this approach can be used to achieve higher levels of server consolidation than currently possible. Overall, as future server environments gravitate towards more memory-constrained and cost-conscious solutions, we believe that the memory disaggregation approach we have proposed in the paper is likely to be a key part of future system designs.

7. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their feedback. This work was partially supported by NSF grant CSR-0834403, and an Open Innovation grant from HP. We would also like to acknowledge Andrew Wheeler, John Bockhaus, Eric Anderson, Dean Cookson, Niraj Tolia, Justin Meza, the Exascale Datacenter team and the COTSon team at HP Labs, and Norm Jouppi for their support and useful comments.

8. REFERENCES

[1] K. Asanovic et al. The Landscape of Parallel Computing Research: A View from Berkeley. UC Berkeley EECS Tech Report UCB/EECS-2006-183, Dec. 2006.

[2] VMWare Performance Team Blogs. Ten Reasons Why Oracle Databases Run Best on VMWare "Scale up with Large Memory." http://tinyurl.com/cudjuy

[3] J. Larus. Spending Moore's Dividend. Microsoft Tech Report MSR-TR-2008-69, May 2008

[4] SIA. International Technology Roadmap for Semiconductors 2007 Edition, 2007.

[5] HP. Memory technology evolution: an overview of system memory technologies. http://tinyurl.com/ctfjs2

[6] A. Lebeck, X. Fan, H. Zheng and C. Ellis. Power Aware Page Allocation. In Proc. of the 9th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), Nov. 2000.

[7] V. Pandey, W. Jiang, Y. Zhou and R. Bianchini. DMA-Aware Memory Energy Conservation. In Proc. of the 12th Int. Sym. on High-Performance Computer Architecture (HPCA-12), 2006

[8] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge and S. Reinhardt. Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments. In Proc. of the 35th Int. Sym. on Computer Architecture (ISCA-35), June 2008

[9] P. Ranganathan and N. Jouppi. Enterprise IT Trends and Implications for Architecture Research. In Proc. of the 11th Int. Sym. on High-Performance Computer Architecture (HPCA-11), 2005.

[10] http://apotheca.hpl.hp.com/pub/datasets/animation-bear/

[11] L. Barroso, J. Dean and U. Hoelzle. Web Search for a Planet: The Google Cluster Architecture. IEEE Micro, 23(2), March/April 2003.

[12] E. Felten and J. Zahorjan. Issues in the implementation of a remote memory paging system. University of Washington CSE TR 91-03-09, March 1991.

[13] M. Feeley, W. Morgan, E. Pighin, A. Karlin, H. Levy and C. Thekkath. Implementing global memory management in a workstation cluster. In Proc. of the 15th ACM Sym. on Operating System Principles (SOSP-15), 1995.

[14] M. Flouris and E. Markatos. The network RamDisk: Using remote memory on heterogeneous NOWs. Cluster Computing, Vol. 2, Issue 4, 1999.

[15] M. Dahlin, R. Wang, T. Anderson and D. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In Proc. of the 1st USENIX Sym. of Operating Systems Design and Implementation (OSDI '94), 1994.

[16] M. Hines, L. Lewandowski and K. Gopalan. Anemone: Adaptive Network Memory Engine. Florida State University TR-050128, 2005.

[17] L. Iftode. K. Li and K. Peterson. Memory servers for multicomputers. IEEE Spring COMPCON '93, 1993.

[18] S. Koussih, A. Acharya and S. Setia. Dodo: A user-level system for exploiting idle memory in workstation clusters. In Proc. of the 8th IEEE Int. Sym. on High Performance

Distributed Computing (HPDC-8), 1999.
[19] A. Agarwal et al. The MIT Alewife Machine: Architecture and Performance. In Proc. of the 23rd Int. Sym. on Computer Architecture (ISCA-23), 1995.
[20] D. Lenoski et al. The Stanford DASH Multiprocessor. IEEE Computer, 25(3), Mar. 1992.
[21] E. Hagersten and M. Koster. WildFire–A Scalable Path for SMPs. In Proc. of the 5th Int. Sym. on High-Performance Computer Architecture (HPCA-5), 1999.
[22] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In Proc. of the 25th Int. Sym. on Computer Architecture (ISCA-25), 1997.
[23] W. Bolosky, M. Scott, R. Fitzgerald, R. Fowler and A. Cox. NUMA Policies and their Relationship to Memory Architecture. In Proc. of the 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV), 1991
[24] K. Li and P. Hudak, Memory coherence in shared virtual memory systems. ACM Transactions on Computer Systems (TOCS), 7(4), Nov. 1989.
[25] D. Scales, K. Gharachorloo and C. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In Proc. of the 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII), 1996.
[26] C. Amza et al. TreadMarks: Shared Memory Computing on Networks of Workstations. IEEE Computer, 29(2), 1996.
[27] I. Schoinas, B. Falsafi, A. Lebeck, S. Reinhardt, J. Larus and D. Wood. Fine-grain Access Control for Distributed Shared Memory. In Proc. of the 6th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), 1994.
[28] K Gharachorloo. The Plight of Software Distributed Shared Memory. Invited talk at 1st Workshop on Software Distributed Shared Memory (WSDSM '99), 1999.
[29] ScaleMP. The Versatile SMP™ (vSMP) Architecture and Solutions Based on vSMP Foundation™. White paper at http://www.scalemp.com/prod/technology/how-does-it-work/
[30] F. Douglis. The compression cache: using online compression to extend physical memory. In Proc. of 1993 Winter USENIX Conference, 1993.
[31] M. Ekman and P. Stenström. A Robust Main Memory Compression Scheme. In Proc. of the 32rd Int. Sym. on Computer Architecture (ISCA-32), 2005
[32] Virident. Virident's GreenGateway™ technology and Spansion® EcoRAM. http://www.virident.com/solutions.php
[33] Texas Memory Systems. TMS RamSan-440 Details. http://www.superssd.com/products/ramsan-440/
[34] Intel. Intel Fully Buffered DIMM Specification Addendum. http://www.intel.com/technology/memory/FBDIMM/spec/Intel_FBD_Spec_Addendum_rev_p9.pdf
[35] T. Kgil et al. PicoServer: using 3D stacking technology to enable a compact energy efficient chip multiprocessor. In Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII), 2006.
[36] M. Ekman and P. Stenstrom. A Cost-Effective Main Memory Organization for Future Servers. In Proc. of the 19th Int. Parallel and Distributed Processing Symposium, 2005.
[37] C. Waldspurger. Memory Resource Management in VMware ESX Server. In Proc. of the 5th USENIX Sym. on Operating System Design and Implementation (OSDI '02), 2002.
[38] D. Ye, A . Pavuluri, C. Waldspurger, B. Tsang, B. Rychlik and S. Woo. Prototyping a Hybrid Main Memory Using a Virtual Machine Monitor. In Proc. of the 26th Int. Conf. on Computer Design (ICCD), 2008.
[39] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero and D. Ortega. COTSon: Infrastructure for System-Level Simulation. ACM Operating Systems Review 43(1), 2009.
[40] J.R. Santos, Y. Turner, G. Janakiraman and I. Pratt. Bridging the gap between software and hardware techniques for I/O virtualization. USENIX Annual Technical Conference, 2008.
[41] J. Rolia, A. Andrzejak and M. Arlitt. Automating Enterprise Application Placement in Resource Utilities. 14th IFIP/IEEE Int. Workshop on Distributed Systems: Operations and Management, DSOM 2003.
[42] R. Bryant and J. Hawkes. Linux® Scalability for Large NUMA Systems. In Proc. of Ottowa Linux Symposium 2003, July 2003.
[43] T. Kgil, D. Roberts and T. Mudge. Improving NAND Flash Based Disk Caches. In Proc. of the 35th Int. Sym. on Computer Architecture (ISCA-35), June 2008.