A HyperTransport-Enabled Global Memory Model For Improved Memory Efficiency

Jeffrey Young, Sudhakar Yalamanchili
Georgia Institute of Technology
jyoung9@gatech.edu, *sudha@ece.gatech.edu*

Federico Silla, Jose Duato
Universidad Politecnica de Valencia, Spain
{*fsilla, jduato*}*@disca.upv.es*

Abstract

Modern data centers are presenting unprecedented demands in terms of cost and energy consumption, far outpacing architectural advances. Consequently, blade designs exhibit significant cost and power inefficiencies, particularly in the memory system. We propose a HyperTransport-enabled solution called the Dynamic Partitioned Global Address Space (DPGAS) model for seamless, efficient sharing of memory across blades in a data center, leading to significant power and cost savings. This paper presents the DPGAS model, describes HyperTransport-based hardware support for the model, and assesses this model's power and cost impact on memory intensive applications. Overall, we find that cost savings can range from 4% to 26% with power reductions ranging from 2% to 25% across a variety of fixed application configurations using server consolidation and memory throttling. The HyperTransport implementation enables these savings with an additional node latency cost of 1,690 ns latency per remote 64 byte cache line access across the blade-to-blade interconnect.

1. Introduction

The current solution to satisfying increasing demand for memory on a blade server is to provision memory on each blade for the worst case demand. One recent study empirically measured memory footprints from non-virtualized applications across 3,000 servers under normal applications and found the average physical memory usage to be about 1 Gigabyte [3]. However, this study also found that memory requirements can vary greatly, with 50% of the applications requiring between 1 GB and 4 GB of memory at certain points during the five-week period of data collection. Thus, provisioning blade memory for the average case can prove to be inadequate with respect to the subsequent page fault rate while provisioning for the worst-case memory footprint can lead to servers that are substantially overprovisioned and consequently expensive and power inefficient. Furthermore, the cost of DRAM is a non-linear function of density and memory size, thus small increases in provisioned memory lead to disproportionate increases in cost.

We hypothesize that while memory demands of individual applications can vary substantially, rarely, if ever, do all applications make peak demands concurrently. The idea proposed by this work is to reduce the cost and power associated with memory by provisioning blades with less than worst-case memory demand and sharing memory across blades during periods of localized, high memory demand. Thus, the physical memory accessible to a blade can vary over time, increasing during periods of peak load by "borrowing" physical memory from an adjacent blade. This idea of shared memory is clearly not new. However, memory sharing via traditional means can exact significant performance penalties through the interconnect and operating system management functions rendering them infeasible in commodity server configurations.

What has changed is the recent introduction of fast interconnects integrated onto the multi-core die close to the memory controllers. The advent of HyperTransport technology reduced the distance from the "wire" to the on-chip memory controller providing low-latency access to remote memory controllers. Thus the hardware cost to access remote memory, e.g., adjacent blades, is no longer prohibitive. However, to productively harness this raw capability, a global system model must be defined to direct how the system-wide memory is allocated/accessed and thereby shared across the operating system domains of distinct blades. This is where our approach differs from prior non-uniform memory access (NUMA) architectures. Each blade is under the control of a distinct OS. However a blade may periodically become a NUMA machine that has access to a portion of the physical memory of an adjacent blade. The ad-

vent of on-die integrated HT makes this feasible from a performance perspective.

This paper proposes a dynamic global address space model (DPGAS) by modifying the existing partitioned global address space model (PGAS) [4] to support a global, *noncoherent physical address space* where an application's virtual address space can be dynamically allocated physical memory located on local and remote nodes. Architectural support for address space management is tightly integrated into the HyperTransport interface to minimize the performance overhead of remote memory accesses and to permit fast, dynamic changes in physical address space mappings. Physical memory is dynamically shared by *spilling* memory demand on a blade to neighboring blades as necessary during peak periods. Consequently, the total amount of memory to be provisioned across the data center can be significantly reduced, leading to substantial cost and power savings with minimal loss of performance (an increase in the page fault rate).

Specifically, this paper contributes the following:

1. A physical address space model, Dynamic Partitioned Global Address Space (DPGAS), for managing system-wide physical memory in large-scale server systems.

2. Design, implementation, and evaluation of hardware support for the DPGAS model via a memory mapping unit that is integrated with a HyperTransport local interface and tunnels memory requests via commodity interconnect—in this case Ethernet.

3. An evaluation of DPGAS with 1) traces from memory-intensive applications, 2) an on-demand memory spilling policy to allocate off-blade memory when local demand exceeds available physical memory, and 3) an evaluation of the cost and power savings from more efficient DRAM usage.

The following sections present the model, its architectural support integrated into the HT interface, and a simulation-based evaluation of the potential for cost and power savings.

2. A Dynamic Partitioned Global Address Space model

The DPGAS model is a generalization of the partitioned global address space (PGAS) model to permit flexible, dynamic management of a physical address space at the hardware level—the virtual address space of a process is mapped to physical memory that can span multiple (across blades) memory controllers. The two main components of the DPGAS model are the architecture model and the memory model.

2.1. Architecture model

Future high-end systems are anticipated to be composed of multi-core processors that access a distributed global 64-bit physical address space. Cores nominally have dedicated L1 caches for instructions and data, but may share additional levels of cache amongst themselves in groups of two cores, four cores, etc. A set of cores on a chip will share one or more memory controllers and low-latency link interfaces integrated onto the die such as HyperTransport [15]. All of the cores also will share access to a memory management function that will examine a physical address and route this request (read or write) to the correct memory controller—either local or remote. For example, in the current-generation Opteron systems, such a memory management function resides in the System Request Interface (SRI), which is integrated on-chip with the Northbridge [6].

2.2. Memory model

The memory model is that of a 64-bit partitioned global physical address space. Each partition corresponds to a contiguous physical memory region controlled by a single memory controller, where all partitions are assumed to be of the same size. For example, in the Opteron (prior to Barcelona core), partitions are 1 TB corresponding to the 40-bit Opteron physical address. Thus, a system can have 2^{24} partitions with a physical address space of 2^{40} bytes for each partition. Although large local partitions would be desirable for many applications, such as databases, there are non-intuitive tradeoffs between partition size, network diameter, and end-to-end latency that may motivate smaller partitions. Further, smaller partitions may occur due to packaging constraints. For example, the amount of memory attached to an FPGA or GPU accelerator via a single memory controller is typically far less than 1 TB. Thus, the DPGAS model incorporates a view of the system as a network of memory controllers accessed from cores, accelerators, and I/O devices.

Two classes of memory operations can be generated by a local core: 1) *load/store* operations that are issued by cores to their local partition and are serviced per specified core-semantics, and 2) *get/put* operations that correspond to one-sided read/write operations on memory locations in remote partitions [22].

Coherence is separated from the issues central to

defining the DPGAS model because large, scalable coherence is still an unsolved research problem, and many systems do not require full-scale coherence across large numbers of servers. Additionally, coherence can be enforced between the one to eight Opteron-based sockets on a server blade to provide local "islands" of coherence. In this case one can view the DPGAS model as dynamically increasing the size of physical memory (across blades) that is associated with a coherence domain although the specific protocols are beyond the scope of this paper.

A sample get transaction on a memory location in a remote partition must be forwarded over some sort of network to the target memory controller and a read response is transmitted back over the same network. The specific network is not germane to the DPGAS model implementation. However, being constrained by commodity parts, this study utilizes Gigabit Ethernet.

Once the DPGAS memory model is enabled, an application's (or process's) virtual address space can be allocated a physical address space that may span multiple partitions (memory controllers), i.e., local and remote partitions. The set of physical pages allocated to a process can be static (compile-time) or dynamic (run-time). Multiple physical address spaces can be overlapped to facilitate sharing and communication.

This paper is only concerned with a very specific application of DPGAS, namely sharing of memory across blades. Dynamic memory requests at a blade can be satisfied by *spilling*—allocating memory from a neighboring blade with spare capacity. We demonstrate in section 5 that this simple allocation policy can have a significant impact. The following section addresses the feasibility of a hardware implementation.

3. DPGAS: implementation

Hardware support for DPGAS has two basic components. The first is a memory function that distinguishes between local and remote memory requests. The second is a memory mapping unit that maps remote physical addresses to specific destination memory controllers. The former is available in modern processors such as the Opteron. The latter is contributed by this paper and is tightly integrated into the HyperTransport interface as shown in Figure 1. The proposed memory mapping unit or bridge performs several functions, including 1) managing remote accesses, 2) encapsulating remote requests into an inter-blade communication fabric (the demonstrator uses Ethernet), and 3) extending noncoherent HT packet semantics across nodes. This section describes the design and implementation of the bridge.

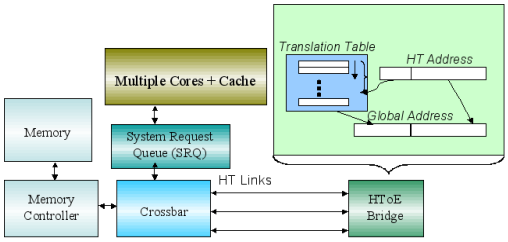


Figure 1. HToE Bridge with Opteron Memory Subsystem

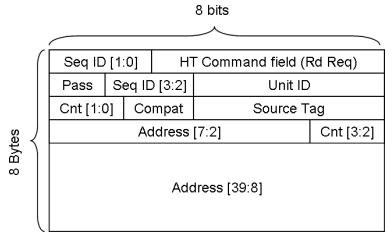
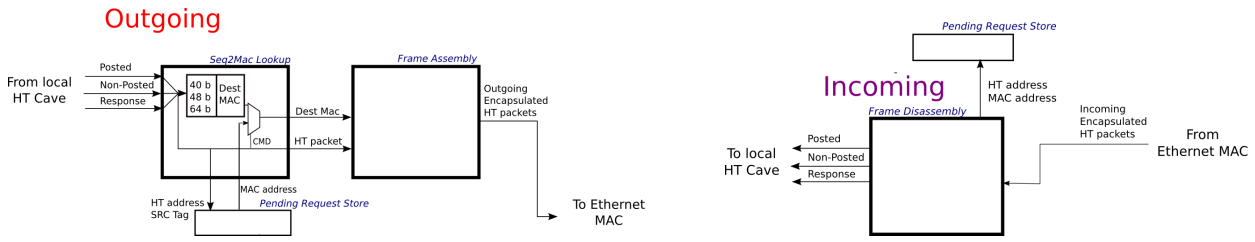


Figure 2. HT read request packet format

3.1. HyperTransport overview

HT is a point-to-point packet switched interconnect standard [15] that defines features of message-based communication, including 1) the use of groups of virtual channels, 2) read/write transactions with posted and non-posted semantics, 3) naming and tracking of multiple outstanding transactions from a source, and 4) specification of ordering constraints between messages. In addition, the HT specification defines flush and fence commands to manage updates to memory on a node. Our model extends the flush command to a remote version while conforming to normal HT ordering and deadlock avoidance protocols.

A typical command packet is shown in Figure 2, where the fields specify options for the read transaction and preservation of ordering and deadlock freedom. Our implementation specifically relies on the UnitID, SrcTag, SeqID, and address fields. The UnitID specifies the source or destination device and allows the local host bridge to direct requests/responses. The SrcTag and SeqID are used to specify ordering constraints between requests from a device, for example, ordering between outstanding, distinct transactions. Finally, the address field is used to access memory that is mapped to either main memory or HT-connected devices. An extended HT packet can be used that builds on this format to specify 64-bit addresses [15].



(a) Outgoing datapath

(b) Incoming datapath

Figure 3. HToE Bridge Components

3.2. HyperTransport over Ethernet—address translation and Ethernet encapsulation

Our demonstrator is based on the use of Ethernet as the commodity inter-blade interconnect primarily due to ready availability of hardware implementations. The bridge design itself does not rely on Ethernet and is easily replaced with other commodity or specialized interconnects. We refer to this demonstrator bridge as the *HT-over-Ethernet (HToE)* implementation. The HToE bridge implementation uses the University of Heidelberg's HyperTransport Verilog implementation [25], which implements an noncoherent HT cave (endpoint) device. Our bridge interfaces with the Heidelberg core so that we can demonstrate functionality with a realistic HT cave implementation. Figures 3(a) and 3(b) show the outbound and inbound components of our HToE bridge along with interface signals for the Heidelberg core and Ethernet MAC.

The HToE implementation is based on a system with Opteron nodes where each Opteron node has an Ethernet-enabled FPGA card available in the HTX connector slot, such as the University of Heidelberg HTX card [2]. Several nodes are connected via an inexpensive Ethernet switch, and it is assumed that HyperTransport messages sent to remote addresses via the HToE bridge are routed using one of two methods: 1) access to the northbridge address mapping tables (via the BIOS) in order to specify the physical address space mappings for the HToE bridge device, or 2) an intelligent MMU that distinguishes between accesses to the local memory and the I/O address space and HT packets that are sent for non-local addresses through the HToE bridge.

Consider a system that has been properly initialized and consider an application that generates a read operation to an address that is in a remote partition. There are three stages in each individual communication operation (e.g., a read request command) at a given source host and attached devices: 1) extension from the 40-bit physical address in the Opteron to the 64-bit physical address, 2) creation of a HT packet that includes a 64-bit extended address, and 3) mapping the most significant 24 bits in the destination address to a 48-bit MAC address and encapsulation into an Ethernet frame. An efficient implementation could pipeline the stages to minimize latency, but retaining the three stages has the following advantages: 1) It separates the issues due to current processor core addressing limitations from the rest of the system, which will offer a clean, global shared address space, thus allowing implementations with other true 64-bit processors, and 2) it will be easy to port to other platforms that do not encapsulate by using Ethernet frames, but use other link layer formats such as Infiniband. Thus, some efficiency was sacrificed for initial ease of implementation and for a cleaner, modular design.

First, the HT packet type is decoded into a request or response command packet in the module called Seq2Mac in Figure 3(a). For request packets the two most significant bits of the 40-bit address are decoded to select one of four partition registers to access the 24-bit partition address—the two most significant bits in the 40-bit address used to address the partition register are reset in parallel with the access to the partition register. Now three pieces of information are needed: 1) the extended 24-bit address to form an HT read request packet with extended address, 2) the MAC address of the destination bridge to encapsulate the extended HT packet into Ethernet, and 3) the local MAC address, according to Ethernet frame format to enable the response. Item 3 has been set during initialization, and access to the source MAC address is not in the critical path. Items 1 and 2 have a direct correspondence among them—given a destination node ID or the remote partition address, there is a unique MAC address associated with both data fields. Therefore, the partition register can store both the 24-bit partition address and the destination MAC address together, thus reducing access time when forming the Ethernet frame. Once the remote MAC address and the 64-bit address have been found in the partition table, the new HT packet is constructed and encapsulated in a standard Ethernet packet, illustrated in the figure

as the Ethernet Frame Assembly module. The encapsulated packet is then buffered until it can be sent using the local node's Ethernet MAC and the physical Ethernet interface. For packets that send a set amount of data, the control and data packets must be buffered until all the data has been encapsulated into Ethernet frames.

The receive behavior of the bridge on the remote node will require a "response matching" table where it will store, for every non-posted HT request (request that requires a response), all the information required to route the response back to the source when it arrives. This table is required since HT is strictly a local interconnect and response packets have no notion of a destination 40-bit (or extended 64-bit) address. Since the formats of HT request and response packets differ and this implementation desires not to change local HT operation, the SrcTag field of each packet is used to match MAC addresses from an incoming request packet with an outgoing response packet. Note that each request packet contains the source MAC address, and this is the address stored in the "response matching" table and later used as the destination MAC address for the corresponding response. Encapsulation and buffering occur once again until the response and data can be transmitted over Ethernet. In the HToE bridge, this module is listed as the Pending Request Store in Figure 3(b) and is shared between incoming and outgoing packets.

It should also be noted that since HT SrcTags are 5 bits, a maximum of 32 outstanding requests can be handled concurrently using the Pending Request Store. This limitation means that additional requests must be queued in the bridge until space is free in the Pending Request Store. If two request packets arrive with the same SrcTag, then the latter packet is remapped before being stored in the table. When the corresponding response leaves the HToE bridge, the SrcTag is mapped back to its original value to ensure proper HT routing on the requesting local node. Once the response reaches the local HToE bridge that initiated the read request, the HT packet is removed from its Ethernet encapsulation. The UnitID is changed again to that of the local host bridge and the bridge bit is set to send the packet upstream. This allows the local host bridge to route responses to the originating HT device. Other transactions, such as a posted write or a non-posted write, involve similar sequences of events. The differences in these transactions are that for posted writes, no data is stored to create a response; for non-posted writes, only a "TargetDone" response is returned and no data needs to be buffered before the response is sent over Ethernet. Similarly, atomic Read Modify Write commands can be treated as non-posted write commands for the purposes of this model.

Table 1. Latency results for HToE bridge

DPGAS operation	Latency (ns)
Heidelberg HT Core (input)	55
Heidelberg HT Core (output)	35
HToE Bridge Read (no data)	24
HToE Bridge Response (8 B data)	32
HToE Bridge Write (8 B data)	32
Total Read (64 B)	1692
Total Write (8 B)	944

4. DPGAS: evaluation of hardware support

Memory mapping is on the critical path for remote accesses. This section reports on the evaluation of a hardware implementation of DPGAS support, the bridge, and the integration into the HyperTransport interface and remote extensions to the HyperTransport protocol required to support DPGAS.

4.1. Bridge implementation

Xilinx's ISE tool was used to synthesize, map, and place and route the HToE Verilog design for a Virtex 4 FX140 FPGA. Synthesis tests using Xilinx software have indicated that the four major modules that make up the bridge are individually capable of speeds in excess of 160 MHz—combined, unoptimized results indicate that the HT bridge is more than capable of feeding a 1 Gbps or faster Ethernet adapter with a 125 MHz clock speed. Evaluations for each of the request and reply critical paths suggest that the latency overhead of the bridge is on the order of 24 to 72 ns (for a control packet with no data and a read request response with eight doublewords of data, respectively). In a Xilinx Virtex 4 FX140 FPGA, an unoptimized placement of the bridge uses approximately 1,300 to 1,500 slices, or approximately 5% to 6% of the chip. Overheads that reduced performance included the use of a serial Gigabit Ethernet MAC interface and the use of only one pipeline to handle packets for each of the three available virtual channels. The latency results for our bridge, the Heidelberg core (used to interface with our bridge) [25], and total latency for the entire path from local to remote memory are listed in Table 1. The bridge latency numbers assume a 125 MHz clock and discount any serialization latency normally associated with Xilinx Ethernet MAC interfaces.

4.2. Bridge and memory subsystem latencies

While our synthesis results proved that the HToE bridge is low-latency, it is also important to understand the overall latency penalty that the memory subsystem contributes to remote memory accesses. The latency values for the HToE bridge component and related Ethernet and memory subsystem components were obtained from statistics from other studies [6] [25] [17] and from the above place and route timing statistics for our bridge implementation. An overview is presented in Table 2. Our HToE implementation was based on a 1 Gbps Ethernet MAC included with the Virtex 4 FPGA, but latency numbers were not available for this IP core. 10 Gbps Ethernet numbers are shown in this table to demonstrate the expected performance with known latency numbers for newer Ethernet standards.

Table 2. Latency numbers used for evaluation of performance penalties

Interconnect	Latency (ns)
AMD Northbridge	40
CPU to on-chip memory	60
Heidelberg HT Cave Device	35 - 55
HToE Bridge	24 - 72
10 Gbps Ethernet MAC	500
10 Gbps Ethernet Switch	200

Utilizing the values from Tables 1 and 2 for using the HToE bridge to send a request to remote memory, the performance penalty of remote memory access can be calculated using the formula:

$$t_{rem\_req} = t_{northbridge} + t_{HToE} + t_{MAC} + t_{transmit}$$

where the remote request latency is equal to the time for an AMD northbridge request to DRAM, the DPGAS bridge latency (including the Heidelberg HT interface core latency), and the Ethernet MAC encapsulation and transmission latency. This general form can be used to determine the latency of a read request that receives a response:

$$t_{rem\_read\_req} = 2{*}t_{HToE\_req}\ 2{*}t_{HToE\_resp} + 2{*}t_{MAC} + 2{*}t_{transmit} + t_{northbridge} + t_{rem\_mem\_access}$$

These latency penalties compare favorably to other technologies, including the 10 Gbps cut-through latency for a switch, which is currently 200 ns [23]; the fastest MPI latency, which is 1.2 μs [21]; and disk latency, which is on the order of 6 to 13 ms for hard drives such as those in one of the server configurations used below for the evaluation of DPGAS memory sharing [26]. Additionally, this unoptimized version of the HToE bridge is fast enough to feed a 1 Gbps Ethernet MAC without any delay due to encapsulating packets. Likely improvements for a 10 Gbps-comptable version of the HToE bridge would include multiple pipelines to allow processing of packets from different virtual channels and the buffering of packets destined for the same destination in order to reduce the overhead of sending just one HT packet in each Ethernet packet in the current version.

5. DPGAS: evaluation of memory sharing

In the absence of a full hardware testbed, we employ a trace-driven analysis of the potential savings offered by a DPGAS implementation. Virtual address traces were acquired using an instrumented SIMICS 3.0.31 model [20] and fed through an internally developed C++ page table simulator to determine the number of page faults as a function of physical memory footprints ranging from 32 MB to 1 GB. Five benchmarks were selected: Spec CPU 2006's MCF, MILC, and LBM [11]; the HPCS SSCA graph benchmark [1]; and the DIS Transitive Closure benchmark [7]. These benchmarks had maximum memory footprints ranging from 275 MB to 1600 MB. A 2.1 billion address trace (with 100 million addresses to warm the page table) was sampled from memory intensive program regions of each benchmark.

5.1. Memory allocation

We analyze the impact of DPGAS by simulating a workload allocation across a multiblade server configuration using a simple greedy bin packing algorithm. An application is randomly selected and its maximum memory footprint is allocated on a random blade. This process is repeated until some termination criterion is met, e.g., allocation failure, fixed workload, etc. The workload is recorded and the same set of memory footprints is allocated across the same server configuration using DPGAS as follows. When an application cannot be allocated on a blade due to a lack of memory, additional memory is allocated on an adjacent blade, i.e., *spilling* the memory request. This is repeated until all application footprints have been allocated.

Two HP Proliant server configurations were selected for analysis, representing high-end and low-end performance points. Both configurations are expected to execute at least two instances of a benchmark application trace per core. These server configurations are detailed in Table 3. All associated system and memory costs and power statistics were derived from [13] and [14].

Table 3. HP Proliant server configurations

Model (HP)	CPU Cores (Opterons)	Max. Memory	Base Cost/Power
DL785 G5	8 quad-core 2.4 GHz	512 GB	~$42,000/1110 W
DL165 G5	2 quad-core 2.1 GHz	64 GB	~$2,000/197 W

5.2. Cost and power evaluation

Results from two classes of experiments are shown here based on results from experiments (as described in Section 5.1) averaged over 50 iterations.

5.2.1. Fixed workload and scale out. These experiments considered fixed workloads where a workload is a fixed number of applications. We based our workload model results from Intel's study of candidate applications for virtualization [3] where an average number of applications per core were identified. We extrapolated this number to a data center with 250 servers (which translates to 2,000 or 500 processor sockets for our server configurations) that could support either 19,500 applications using high-end servers or 4,700 applications using low-end servers. Additionally, we investigated the effects of scaling the number of blades while keeping the workload fixed.

This set of experiments used a baseline configuration with a fixed 64 GB of memory per blade and standard bin packing allocation where application memory footprint had to reside within a blade. The resulting fragmentation left unused memory across blades although several blades exhibited very high memory utilization (in excess of 60 GB). For comparison purposes, we considered a DPGAS-enabled server configuration where half the blades were provisioned with 64 GB and half with less memory. The aggregate difference in memory is roughly equal to the unutilized memory in the first configuration. This latter configuration corresponds to a data center with half of the servers overprovisioned (receivers in our model) and half of the servers minimally provisioned (spill memory to other nodes). Finally we repeated the experiment with 56 GB per blade rather than 64 GB, which reduced memory fragmentation.

The total cost savings for the low- and high-end server configurations are shown in Figures 4 and 5 with savings between standard and DPGAS allocation graphed as the third column of each group. As we see in the base (250-server) case, DPGAS has the potential to save 15% to 26% in memory cost when the initial provisioning is high (64 GB), which translates into a $30,736 savings for the low-end servers and $200,000 for the high-end servers. On the other hand, with lower initial

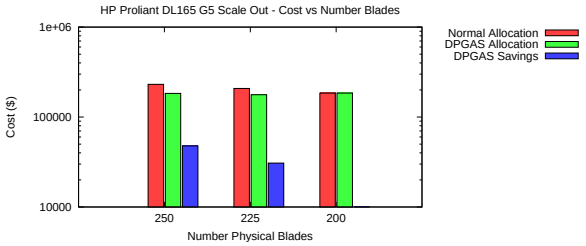


Figure 4. Scale out cost for Proliant DL165 G5

memory (56 GB), the savings in Figure 5 are 13%, or $103,365

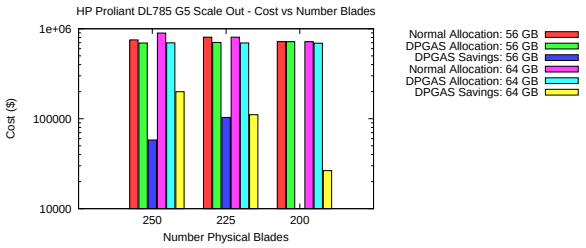


Figure 5. Scale out cost for Proliant DL785 G5

It is also important to notice that savings with DPGAS allocation drops as applications are consolidated onto fewer servers. This is likely due to the fact that there is less fragmentation with no sharing and therefore less inefficiency to be recovered.

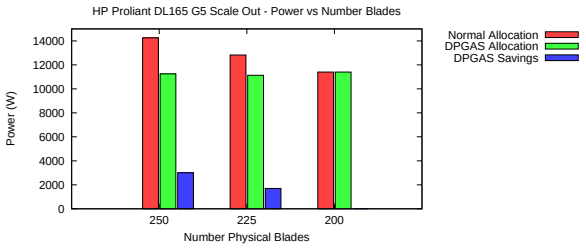


Figure 6. Scale out power for Proliant DL165 G5

Similarly, the power savings using DPGAS allocation (Figures 6 and 7) is substantial in the base case, with savings of 3,625 (25%) and 5,875 (22%) watts of input power for the low-end and high-end server configurations, respectively. When server consolidation onto 200 servers is used, power savings drops substantially to 800 and 500 watts for the same configurations. The smaller memory configuration results for the high-end server also demonstrate smaller savings of 2500 watts in the 250 server case. Both the cost and power results indicate that DPGAS memory allocation is most effective when fragmentation is normally high and when variance in workload memory footprint is high.

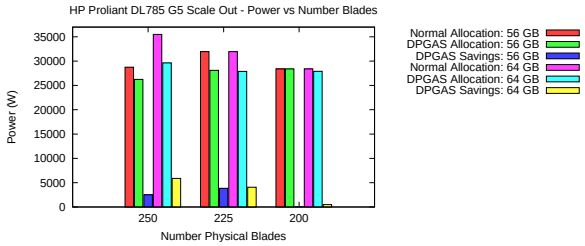


Figure 7. Scale out power for Proliant DL785 G5

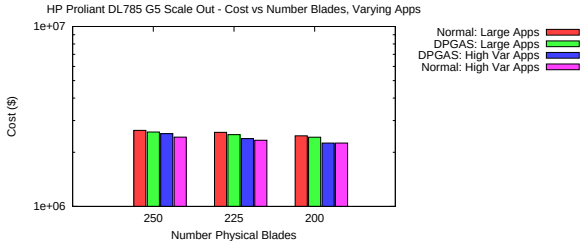


Figure 8. Scale out cost for Proliant DL785 G5 - varying workload sizes

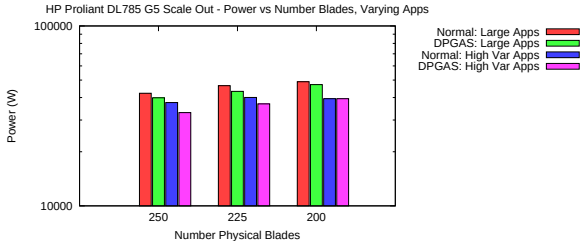


Figure 9. Scale out power for Proliant DL785 G5 - varying workload sizes

To further investigate the effects of memory fragmentation on cost and power, we also ran two separate sets of allocations using workloads drawing from 1) a pool of three applications with large memory footprints and 2) a pool of two applications with small and very large footprints. This experiment included the use of a synthetic benchmark with a memory footprint of 2275 MB that represented a large, unknown enterprise workload similar to those in [3]. The results can be seen in Figures 8 and 9 with cost savings of 2% to 3% for the large applications and 2% to 4% for the second application set. Power savings range from 6% to 7% for large applications and 8% to 12% for the second set of applications. The dropoff in performance can be explained as follows. When application footprints are of similar size, the bin packing behavior of allocation produces little fragmentation, but when applications have small footprints, they can fill unallocated memory and reduce fragmentation. DPGAS seems to work best when the dynamics are such that a wide range of footprints are likely, leading to fragmentation that can be otherwise recovered by DPGAS.

5.2.2. Memory throttling. Memory throttling is where the allocated footprint per application is less than the maximum footprint at the expense of an increased page fault rate. We compared two additional cases with 250 servers: 1) Each server had 50% of the original memory and each application was allocated 50% of its maximum memory footprint, and 2) each server had 25% or the original memory, and each application received 25% of its maximum footprint. The results for cost and power in the high-end server are shown in Figures 10 and 11. The effects of memory throttling are significant. For instance, reducing memory from 64 GB to 32 GB in each server reduces memory cost by $478,000 and memory power by 17,750 watts (from a base cost of $897,000 and base power of 35,500 watts). The usage of DPGAS allocation with 50% memory throttling with the high-end server configuration can reduce the total memory cost by $570,000 and total memory power by 21,125 watts.

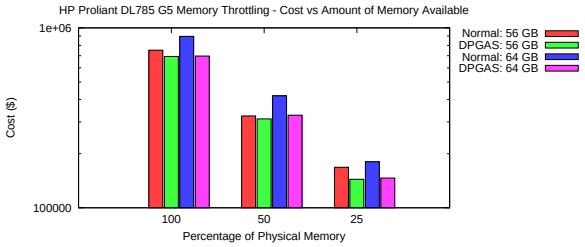


Figure 10. Memory throttling cost for Proliant DL785 G5

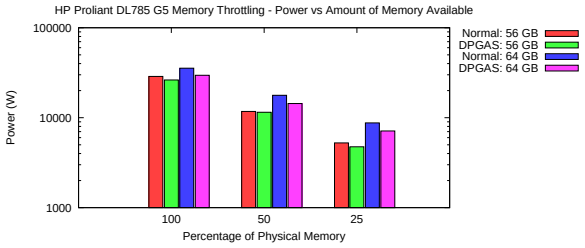


Figure 11. Memory throttling power for Proliant DL785 G5

At the lower bound of savings, reducing memory in the high-end server from 56 GB to 28 GB or 14 GB results in similar savings due to memory throttling, but the savings from using DPGAS is somewhat lower with cost savings of 4% to 14% and power savings of 2% to 10%. This translates to cost savings of $12,000 to

Table 4. HP Proliant 165 G5 cost and power with memory throttling

Allocation	No Throttling	50% Throttling	25% Throttling
Normal ($)	$230,750	$111,250	$51,500
DPGAS ($)	$183,000	$97,125	$51,500
Normal (W)	14,250	7,000	3,500
DPGAS (W)	11,250	5,875	3,500

$24,000 over the normal case and power savings of 250 to 500 W, using 50% and 25% memory throttling.

Additional statistics for the low-end server configuration are shown in Table 4. These experimental results concur with the high-end server configuration, except that power and cost savings are smaller due to less memory fragmentation and less memory overall for remote sharing. In the 25% memory throttling case, there is not enough leftover memory to be utilized with DPGAS, so no savings are incurred. Overall, DPGAS enables a 4% to 22% reduction in memory cost and a 2% to 25% reduction in memory power when compared to normal allocation for both the low- and high-end servers.

When using memory throttling, performance must also be taken into account. The results from our trace-driven analysis of the benchmark applications provide data on page fault rates that directly correspond to the amount of memory a benchmark is allocated. These results are used to generate Figures 12 and 13 that demonstrate the effects of memory throttling on random allocations of each of our benchmark applications. In general, the usage of memory throttling leads to an order-of magnitude increase in the number of page faults for all applications, but some applications with small memory footprints or random access patterns (poor spatial reuse) are affected much more by using memory throttling with normal allocation.

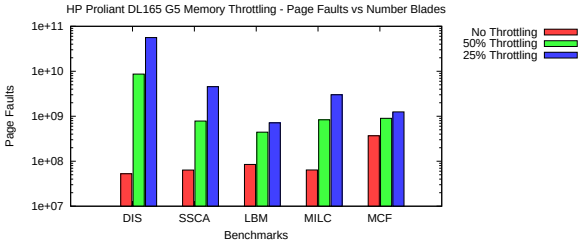


Figure 12. Memory throttling performance for Proliant DL165 G5

6. Related Work

Other researchers have also been focused on the growing power and cost implications of large clusters and server farms. Feng, et al. [5] discussed the efficiencies associated with large servers and proposed

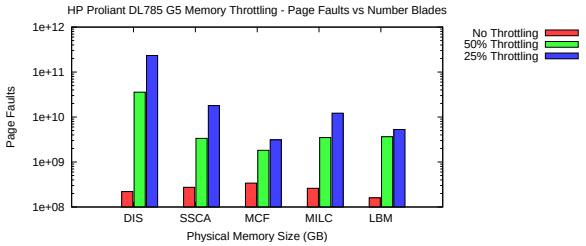


Figure 13. Memory throttling performance for Proliant DL785 G5

a power-efficient supercomputer called Green Destiny. Other strategies have included dynamic voltage scaling for power-aware computing [10] with a focus on CPU power. Raganathy, et al. [24] has also suggested that power-management should take place at the server enclosure levels so that individual systems are not over-provisioned. This study also focused mainly on high-level CPU power management, not memory power.

However, Lefurgy's 2003 study [18] cited important reasoning behind why DRAM cost and power should be considered as a major component in improving overall server efficiencies. Several other researchers have also begun focusing on memory power management at the architecture level, including [16], which proposes using adaptive power-based scheduling in the memory controller, and [9], which uses power "shifting" driven by a global power manager to reduce power of the overall system based on runtime applications.

At the operating system level, [12] proposed a power-aware paging method that utilizes fast MRAM to provide power and performance benefits. Tolentino [27] also suggested a software-driven mechanism to limit application working sets at the operating system level and reduce the need for DRAM overprovisioning.

An evaluation of power and cost trends similar to the ones in this paper was conducted in [19], concluding that separate PCI Express-based memory blades could be used to reduce overall memory usage and memory cost and power. [8] investigated real-world statistics for some of the large "warehouse-sized" server farms that Google runs.

7. Conclusion

With increasing server power and cost outpacing related performance gains, a focus on making data centers and clusters as efficient as possible is vital from a business perspective. We present a new address space model, the Dynamic Partitioned Global Address Space, and define an associated dynamic hardware-based address translation scheme for efficiently utilizing remote

memory with low-latency interconnects such as HyperTransport. An implementation of this model has been developed by encapsulating HyperTransport packets in Gigabit Ethernet via our HT over Ethernet bridge, and initial synthesis results indicate that remote read and write operations are low-latency and comparable to fast message-passing implementations. The impact of low-latency remote access on the ability to share memory is significant, and future plans include the pursuit of a HW/SW testbed to evaluate a complete solution. Additional future work is described in [28].

References

[1] David A. Bader and Kamesh Madduri. Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. In *HiPC*, pages 465–476, 2005.

[2] Ulrich Bruening. The htx board: The universal htx test platform. http://www.hypertransport.org/members/u_of_man/htx_board_data_sheet_UoH.pdf.

[3] S. Chalal and T. Glasgow. Memory sizing for server virtualization. 2007. http://communities.intel.com/docs/.

[4] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05*, pages 519–538, New York, NY, USA, 2005. ACM.

[5] Wu chun Feng. Making a case for efficient supercomputing. *ACM Queue*, 1(7):54–64, 2003.

[6] Pat Conway and Bill Hughes. The amd opteron northbridge architecture. *IEEE Micro*, 27(2):10–21, 2007.

[7] Dis stressmark suite, updated by uc irvine. 2001. http://www.ics.uci.edu/~amrm/hdu/DIS_Stressmark/DIS_stressmark.html.

[8] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *"ISCA 2007: Proceedings of the 34th annual international symposium on Computer architecture"*, pages 13–23, New York, NY, USA, 2007. ACM.

[9] Wes Felter, Karthick Rajamani, Tom Keller, and Cosmin Rusu. A performance-conserving approach for reducing peak power consumption in server systems. In *ICS '05*, pages 293–302, New York, NY, USA, 2005. ACM.

[10] Rong Ge, Xizhou Feng, and Kirk W. Cameron. Improvement of power-performance efficiency for high-end computing. In *IPDPS '05*, page 233.2, Washington, DC, USA, 2005. IEEE Computer Society.

[11] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[12] Y. Hosogaya, T. Endo, and S. Matsuoka. Performance evaluation of parallel applications on next generation memory architecture with power-aware paging method. *IPDPS '08*, pages 1–8, April 2008.

[13] Hp proliant dl servers - cost specifications. 2008. http://h18004.www1.hp.com/products/servers/platforms/.

[14] Hp power calculator utility: a tool for estimating power requirements for hp proliant rack-mounted systems. 2008. http://h20000.www2.hp.com/bc/docs/support/SupportManual/c00881066/c00881066.pdf.

[15] Hypertransport specification, 3.00c, 2007. http://www.hypertransport.org.

[16] Ibrahim Hur and Calvin Lin. A comprehensive approach to dram power management. In *HPCA '08*, 2008.

[17] Intel 82541er gigabit ethernet controller. http://download.intel.com.

[18] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W. Keller. Energy management for commercial servers. *Computer*, 36(12):39–48, 2003.

[19] Kevin Lim, Parthasarathy Ranganathan, Jichuan Chang, Chandrakant Patel, Trevor Mudge, and Steven Reinhardt. Understanding and designing new server architectures for emerging warehouse-computing environments. In *ISCA '08*, pages 315–326, Washington, DC, USA, 2008. IEEE Computer Society.

[20] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[21] Mellanox connectx ib specification sheet, 2008. http://www.mellanox.com.

[22] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: a portable "shared-memory" programming model for distributed memory computers. In *Supercomputing '94*, pages 340–349, New York, NY, USA, 1994. ACM.

[23] Quadrics qs ten g for hpc interconnect product family. 2008. http://www.quadrics.com/.

[24] Parthasarathy Ranganathan, Phil Leech, David Irwin, and Jeffrey Chase. Ensemble-level power management for dense blade servers. In *ISCA '06*, pages 66–77, Washington, DC, USA, 2006. IEEE Computer Society.

[25] David Slogsnat, Alexander Giese, Mondrian Nüssle, and Ulrich Brüning. An open-source hypertransport core. *ACM Trans. Reconfigurable Technol. Syst.*, 1(3):1–21, 2008.

[26] Storagereview.com drive performance resource center. 2008. http://www.storagereview.com/.

[27] Matthew E. Tolentino, Joseph Turner, and Kirk W. Cameron. Memory-miser: a performance-constrained runtime system for power-scalable clusters. In *CF '07*, pages 237–246, New York, NY, USA, 2007. ACM.

[28] Jeffrey Young, Sudhakar Yalamanchili, Federico Silla, and Jose Duato. A hypertransport-enabled global memory model for improved memory efficiency (tech report), 2008. http://www.cercs.gatech.edu/tech-reports/index08.shtml.