Unleash your Memory-Constrained Applications:

a 32-node Non-coherent Distributed-memory Prototype Cluster

Héctor Montaner*, **Federico Silla***, **Holger Fröning**†, and **José Duato***
**Universitat Politècnica de València, Departament d'Informàtica de Sistemes i Computadors*
Camino de Vera, s/n 46022 Valencia, Spain. hmontaner@gap.upv.es, {fsilla,jduato}@disca.upv.es
†University of Heidelberg, Computer Architecture Group
B6, 26, Building B (3rd floor) 68131 Mannheim, Germany. froening@uni-hd.de

***Abstract*—Improvements in hardware for parallel shared-memory computing usually involve increments in the number of computing cores and in the amount of memory available for a given application. However, many shared-memory applications do not require more computing cores than available in current motherboards because their scalability is bounded to a few tens of parallel threads. Nevertheless, they may still benefit from having more memory resources. Additionally, the performance of extended systems involving more cores is typically constrained by the glueing coherency protocol, whose overhead lowers the performance of the final system.**

In this paper we present a 32-node prototype of a new non-coherent distributed-memory architecture for clusters, aimed to provide applications additional memory borrowed from other nodes without providing them more cores, thus avoiding the penalty of maintaining coherency among nodes of the cluster. Results from the execution of real applications in this prototype demonstrate that our proposal truly works, as well as its performance is assessed.

***Keywords*-Cluster, Hypertransport, memory-hungry, remote memory, memory aggregation.**

I. INTRODUCTION

High performance computing (HPC) has been traditionally addressed by aggressively parallelizing applications and providing them with the hardware that supports such level of concurrency. However, very large scale shared-memory systems have never been feasible due to the overhead introduced by the coherency protocol. Thus, mainframes like the IBM z series [1], featuring a relatively large number of computing cores and up to 2 TB of memory, have traditionally been the largest shared-memory machines. Unfortunately, they are extremely expensive.

On the opposite end we find inexpensive x86-based computers able to scale up to 64 cores [2][3] while providing a few hundred Gigabytes of memory. These machines are currently the building block for clusters, which represent a cheap and powerful choice for HPC. However, clusters do not provide a global shared-memory system. In order to provide a single coherent distributed shared-memory system, some kind of aggregation could be used, like the NumaChip [4], that glues together the available cores and memory resources. However, this aggregation lacks scalability because of the limitations and overhead imposed by the protocol that keeps coherency among the nodes of the cluster.

On the other hand, many applications do not scale beyond the amount of cores located in a single motherboard [5], although they may still benefit from the large memory resources present in those shared-memory clusters. In this case, there is no real need to provide coherency among processors located in different nodes if all the threads from a given application are confined to the processors (and caches) in the same motherboard. If so, there is no reason for propagating coherency operations to caches outside the node that triggers the memory access. Therefore, in this context, aggregating resources like in the NumaChip may be counterproductive because of the overhead of the inter-node coherency protocol. Thus, there is a need for decoupling processor aggregation from memory aggregation.

We have recently presented a new non-coherent distributed-memory architecture for clusters [6] that leverages this observation, thus avoiding the penalty due to the inter-node coherency protocol. Our proposal dynamically partitions the cluster into non-overlapping coherent domains, each of them containing the cores and caches of a single motherboard and potentially spanning to memory located in other motherboards.

Although restricting the amount of cores devoted to an application to those available in a single motherboard may limit the usefulness of our proposal, it is actually very promising given the current and mid-term trends in processor development, motherboard implementations, and parallel programming. The reason is that, on one hand, shared-memory applications do not usually scale beyond a few tens of threads and, on the other hand, current motherboards can allocate up to 64 cores, while in the future this number will probably increase, making our proposal even more appealing.

In [6], the feasibility of the new architecture was analyzed by simulation, showing promising results. In this paper we present a prototype for that architecture. The prototype cluster is able to allocate memory on-demand from other nodes of the cluster to a process running in one of the nodes. Additionally, it also allows to simultaneously distribute all the memory in the cluster among multiple applications running in the same or in different nodes.

The remainder of this paper is organized as follows: in the next section we present a summary of related work. The insights of the proposed architecture are described in Section III. Section IV introduces the new prototype. Section V presents performance results showing the feasibility of our proposal. Finally, Section VI draws some conclusions.

II. Related Work

Disk swapping is the traditional approach for getting additional memory. However, when the working set of an application is bigger than the available physical memory, thrashing increases execution time to prohibitive levels. Remote swap [7] is a similar technique that offloads memory pages to other computers of the cluster, aiming that retrieving those pages from remote memory will be faster than retrieving them from the local hard disk. However, remote swap still suffers from software overhead as each page miss must be handled by the operating system (OS).

A different approach is followed by Violin Memories, that offers a memory server holding up to 504GB of RAM [8]. Unfortunately, the OS is involved in every memory access, which increases access latencies up to 3μs.

Numascale [4], SGI [9], and the extinct 3Leaf [10] provide more resources to applications by aggregating the processors and memory in a cluster into a single computer. Nevertheless, although they are hardware-based approaches, as coherency must be maintained throughout the aggregated computers, scalability and performance in these proposals are limited in practice.

Aggregation could also be achieved by software [11][12]. These approaches rely on libraries, OSes, or Hypervisors to manage a per-page memory migration mechanism and, again, the coherency protocol restricts performance. For example, a single memory reference going through the vSMP ScaleMP layer takes 25μs, while the latency in the high performance ASIC-based SGI approach is 1μs. In our architecture, prototyped using FPGA technology, latency is 2μs as will be shown later.

An orthogonal and complementary approach to the memory limitation problem is to install more memory, especially by using SSD as a cheap substitute for DRAM. In [13] they propose a solution for accelerating data-intensive applications by using flash memory as swap space (a page swap takes 75μs). This extra memory is logically allocated between main memory and the traditional spinning disk, contrary to our architecture where the extra memory is at the same level as main memory, as explained later. On the other hand, as flash technologies are rapidly improving in terms of bandwidth and latency, new studies analyze the potential of using them as main memory [14]. However, with data-intensive applications that proposal does not perform satisfactorily, especially for write operations. Moreover, this approach leads to an imbalanced cluster with an inflexible resource structure, where only the few nodes owning large memory resources are able to execute data-hungry applications, and also to a waste of energy when the extra memory remains idle [15]. Additionally, some researchers are investigating ways to employ emerging memory technologies such as phase-change memory (PCM) and spin torque transfer (SST)-RAM to reduce memory power, but it is unclear whether these will ever reach the market with price, density, and latency characteristics suited to be used as main memory.

III. A New Approach for Shared-Memory in Clusters

We can think of this architecture as a new memory system for clusters. The key factor of our proposal is the way in which processors access memory physically attached to other nodes. This process of accessing remote memory has been designed in order to achieve the lowest possible latency, but bearing in mind that applications should be able to make use of this system automatically.

In order to adhere to the second requirement, any shared memory application should be successfully executed in our system without modifying its code. Actually, as it will be explained next, there is no need for even recompiling the applications because our architecture does not rely on any software library or run-time, so much so that any existing *x86_64* binary will directly make use of our proposal. In this way, the underlying architecture is absolutely transparent to the application, similar to remote swap scenarios and distributed shared memory architectures.

Therefore, if a preexisting binary can be executed in our architecture, the question is how the remote memory accesses are triggered. In a software DSM or, in a more simplistic way, a remote swap system, the event that triggers the remote access mechanism is the page fault. A binary code contains references to virtual addresses that must be translated to physical addresses before the processor forwards these memory requests to the corresponding memory bank. As it is well known, the Memory Management Unit (MMU) is responsible for translating a virtual address by looking up the Translation Lookaside Buffer (TLB), an associative cache memory, or by looking up the Page Table structure in case the TLB has not the requested page translation. We can suppose, without losing generality, that this process is done automatically by hardware. On the other hand, in case the Page Table has not a valid entry for the requested virtual address, a software handler will deal with that page fault by resolving the address and filling the Page Table with the appropriate translation, so that the read operation issued by the processor could continue.

This page fault event is the trigger for the process of accessing remote memory in the software DSM and remote swap scenarios. In the case of remote swap, at this point and before inserting the new translation into the Page Table, the corresponding memory page must be retrieved from remote memory and allocated somewhere in local memory. In the case of software DSM, this memory location must be retrieved from remote memory too, possibly in a more sophisticated way, but again through a similar event-based entry point.

The trigger for accessing remote memory in our proposal does not exist because the remote memory accesses are not treated in any special way. On the contrary, a remote read or write is actually a regular load or store assembly instruction, and it does not need any dedicated handling by the processor or operating system. As a result, latency to remote memory is significantly lower. However, note that the main difference

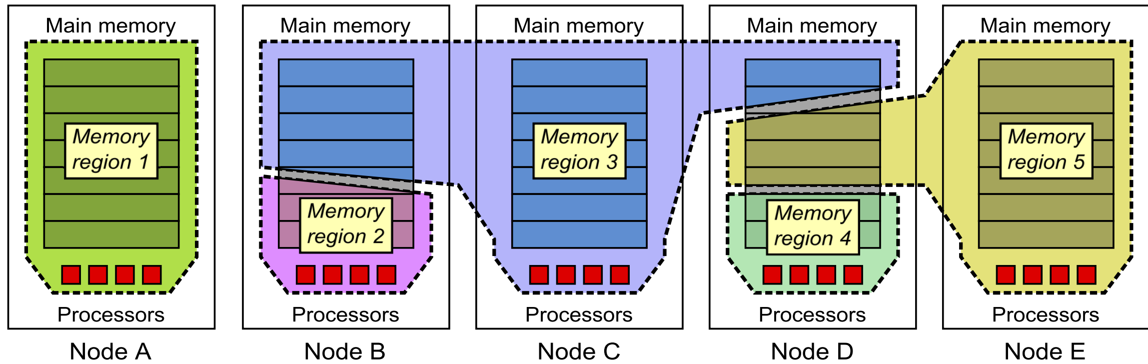


Figure 1. An example of memory sharing among the nodes of a cluster

is not related to latency, but to the nature of the architecture, as our system is not *event-based* and does not involve any software for accessing remote memory.

In this section we present the key component of our proposal: how memory located at other motherboards is efficiently accessed. Nevertheless, deploying the full system requires additional minor components, not described in this paper, such as:

- augmenting the OS kernel so that memory can be progressively hot-plugged and hot-removed as required,
- augmenting the OS services so that knowledge of the location of free memory across the cluster is achieved,
- concerns related to communication reliability and security

A. System Overview

Up to this point, we have seen one of the reasons why our proposal has such a potentially low latency to remote memory: there is no software involved. There is another orthogonal reason: the lack of coherency among motherboards in the cluster. Although the extra latency introduced by the coherency protocol is difficult to assess in terms of absolute numbers (it depends on the exact state of the caches, that is, on the application behavior), traditionally it has been the scale limitation factor in hardware and software shared memory systems, and this is why computers with more than a few hundreds of cores are based on the message passing paradigm.

Keeping coherency when the number of cores goes beyond a few hundreds is prohibitive as it takes too much time to perform some memory operations. In the case of writes, unless the data is in the local cache with the appropriate state, the write operation must cross the system to the corresponding controller, check for permissions and eventually invalidate every copy in the system (by broadcasting to every cache or by relying on a directory located in memory), before the processor can commit the operation. For read operations, the process is similar, and the state of every cache in the system has to be taken into account before the value can be read. We propose to drop coherency in the inter-node space and only maintaining coherency inside each node. This characteristic will certainly prevent the creation of a typical coherent distributed shared-memory space along the cluster, as SGI or Numascale do, but will allow the creation of a *distributed exclusive* memory space. As we mentioned before, it is important to decouple memory aggregation from processor aggregation, and here it is where this feature plays its role.

To better understand our proposal, let us introduce a helpful term: *memory region*. A memory region is an amount of memory made up of one or more logical portions of main memory that could be located at different nodes of the cluster, and that conform altogether a single coherency domain. A process can freely use the entire memory in the region it belongs to but it has no access to the memory in other regions in the cluster. Similarly, a processor can address any location of its memory region, but cannot address memory locations outside it. Figure 1 shows a five-node cluster example. Region 1, confined to node A, represents the regular configuration, that is, processes in that node can access the entire node's memory. On the other hand, region 3 has been extended to the neighbors of node C, so processes in this node have now direct access to part of the memory located in nodes B and D. In this way, regions 2 and 4 have been shrunk and they occupy only a portion of the main memory in nodes B and D, respectively. Moreover, although enlarged memory regions in Figure 1 have spanned to their neighbor nodes, any node may extend its memory resources by borrowing memory from any node in the cluster. Finally, note that the amount of memory allocated to a given region is dynamically adjusted as processes in that region require additionally memory.

In summary, we propose to extend the amount of memory assigned to a process without increasing the number of processors where this process is allowed to run. In this way, an application can use memory physically attached to other nodes in the cluster without involving the caches located in those other nodes, thus, not requiring coherency among cluster nodes. In practice, we have one independent OS at each node and every process is confined to the processors

and caches located at one single node. However, our system allows processes to use memory initially owned by other operating systems at other nodes in the cluster, without incurring in additional coherency traffic across nodes.

The new architecture presents very good scalability: effectively, in our system, the size of a memory region has no impact on the performance of the coherency protocol because the number of caches sharing data in any region is constant. Thus, write operations in a memory region are only notified to the caches of that region. No matter how large the region is, only the caches contained in one node will be informed. As can be seen, a memory-hungry application can be fed with as much memory as present in the cluster without undergoing the coherency overhead.

B. System Implementation

As we said, in order to access memory located in other nodes, our system does not rely on any kind of run-time or communication library and does not require applications to be modified. Actually, applications are not aware that they are making use of remote memory. Additionally, accessing remote memory completely relies on hardware, thus being free of any software overhead. In our proposal, processors do not distinguish between local and remote accesses as both are the result of regular load or store instructions. If the addressed memory turns out to be remote, then the memory access will automatically be forwarded to the corresponding remote node. We accomplish this is by means of HyperTransport.

HyperTransport (HT) [16] is used to interconnect the AMD Opteron processors in a motherboard, where each processor is attached to part of the physical memory by means of its own memory controller. Therefore, as there are several memory controllers, processors require to know where to forward a given memory request. This is achieved by including at each processor a set of address base/limit registers configured at boot time. Hence, when a processor issues a load or store operation related to a given memory location, the processor compares the requested address with those registers, and then forwards the memory access to the memory controller responsible for that memory address. Forwarding the memory operation involves the generation of a HyperTransport packet.

The system described above is the basis upon which we will implement our proposal, which involves creating a new hardware component that we will refer to as *Remote Memory Controller* (RMC). This new component will be presented to the processors in the motherboard as a HyperTransport I/O Unit and will be responsible for memory requests sent from the local system to a remote node. After properly configuring the mentioned address base/limit registers, processors will automatically forward HT transactions to their local RMC, that will forward these requests to the corresponding remote node, where these transactions are handled by the memory controllers of the remote CPUs. Obviously, before accessing remote memory, a reservation phase that assigns remote memory to the process must be carried out. Insights of this process are explained in next section.

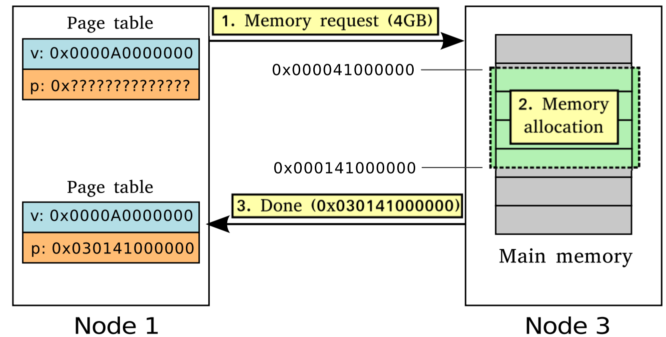


Figure 2. Node 1 reserves remote memory in node 3

C. Remote Memory Reservation Process

When the application needs more memory and the local resources are exhausted, the OS starts the reservation process for getting additional memory from other nodes. In Figure 2 we can see the required steps. First, the OS locates new memory in the cluster. In our example, Node 3 will provide the required memory. After that, Node 1 sends a message requesting 4GB, for instance. At this point, Node 3 actually allocates that memory. Let us assume, for simplicity, that the reservation is done over a contiguous physical memory area. Finally, Node 3 responds to Node 1 with the physical address of the allocated memory area. However, one modification is done to this address before it is sent back: the identifier of the memory owner is coded in the most significant bits of the physical address. In our example, if physical addresses are 48-bit long and there are 256 nodes in the cluster, then we may use the 8 most significant bits to encode the owner and the other 40 bits to specify the address in that node. The key idea is that we can insert in the address itself additional information that will later simplify the access to remote memory.

Once memory has been reserved, when a processor in Node 1 issues a memory operation related to addresses in Node 3, as the most significant bits are not zero (and consequently the data is present in another node), the processor, attending to its base/limit registers, will forward the memory request to the local RMC. The RMC inspects these most significant bits and forwards the request to the corresponding remote RMC, that will set these bits to zero and finally forward the request to a memory controller in that remote node. As can be seen, there is no need for translation tables in the RMC, so minimum functionality has to be implemented in the RMC, and thus small overhead due to message processing is generated.

Note that the memory reservation process is a non-critical process: remote memory can be reserved in advance when the OS realizes that it is running out of memory. Therefore, as this reservation process has no impact on the application execution time, we rely on software to manage the remote

memory reservation instead of increasing the complexity of the hardware.

Finally, it is worth to emphasize that contrary to remote swap and DSM scenarios, where only pages currently in local memory have a translation in the Page Table (remember their event-based nature), in our system every memory page has a translation even if these pages are allocated in remote memory (in essence, remote memory is the same as local memory except for a higher latency) . In this way, as in conventional computers, in our system a page fault exception will only occur the first time the page is accessed and subsequent references to remote memory pages will be entirely managed by hardware as regular memory operations.

IV. A 32-NODE PROTOTYPE

We have built a prototype that implements our proposal for non-coherent distributed-memory. Our prototype consists of 32 nodes based on the Supermicro H8QM8-2+ motherboard containing four 2.1GHz quad-core Opteron processors. Each processor is attached 4GB of 800MHz DDR2 memory. Thus, each node features 16 cores and 16GB of main memory, accounting for a total of 512GB of RAM. Notice, however, that although this is the maximum amount of memory we can currently share in our prototype, nothing except the economical cost of memory prevents us to provide up to 4 TB of memory to an application, as the used motherboards can hold up to 128GB.

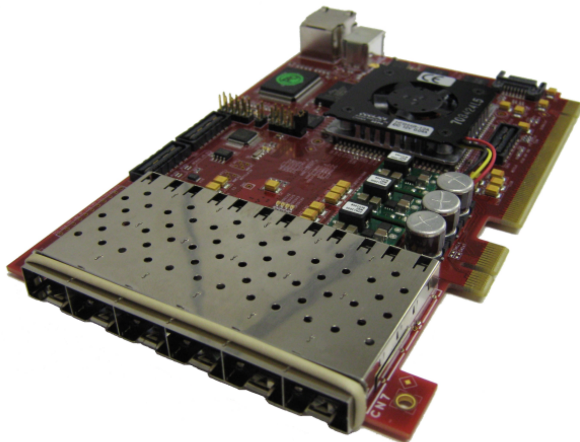


Figure 3. HTX card used to implement the RMC

The motherboards used include an HTX connector[1], where we have attached the card designed by University of Heidelberg [17]. This card includes an FPGA that will be used to implement the RMC functionality. It also includes six fiber links. We will use four of them to interconnect the 32 nodes in a 4x8 2D mesh. The routing functionality will also be implemented in the FPGA which will communicate to the local system through an HT interface running at 400MHz (HT400).

[1]The HTX connector is one of the standard connectors for HT.

The design of the RMC we have developed presents this new component to the rest of the elements in the motherboard as a new HyperTransport memory mapped I/O unit, as mentioned before. The consequences of this implementation is that an Opteron processor can only have one outstanding memory request targeted to the memory region mapped to the RMC. Therefore, when an application intensively accesses remote memory, a new remote memory request cannot be issued before the previous one has been completed. This will reduce overall performance with respect to executing the application using local memory because in this latter case Opteron processors can have up to eight outstanding requests.

V. PERFORMANCE EVALUATION

It is inevitable that this new architecture increases the latency of memory accesses: as the propagation speed of signals is limited, any decrease in spatial locality results in larger access times. Therefore, the goal of this section is to show the feasibility of our proposal by quantifying the slowdown experienced by different applications when using remote memory.

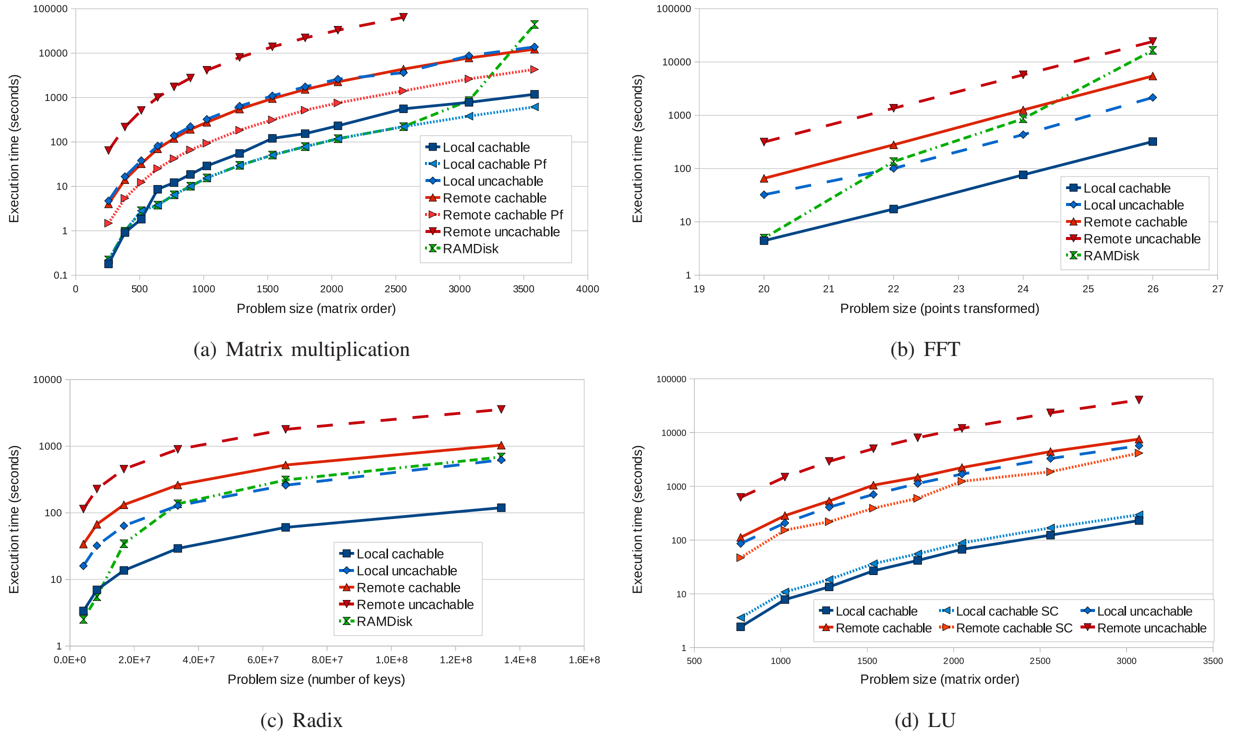
In the following, the term “application node” will refer to the node where the application demanding remote memory is being executed, while the term “memory host” will refer to the node lending memory.

A. Single Memory Host

Figure 4 presents the execution time of four applications in different scenarios and for several problem sizes (memory requirements). The different scenarios basically consist of executing the applications mapping all their data structures either to local memory (lower bound) or to remote memory (upper bound). Moreover, we additionally consider cases with and without caching of data memory accesses. Note that the application code is mapped to local cachable memory in all cases. An additional scenario makes use of a RAMDisk configured as swap space, so that when the application footprint exceeds the available free main memory (set to 100MB), the swap mechanism will move pages from that region to the region of the RAMDisk (also in local main memory, up to 16GB). The purpose of this scenario is to study the overhead of the software layer, which may be representative of a common software DSM solution. Finally, measurements ending with *Pf* or *SC* are optimizations and will be explained in Section V-D.

Figure 4 shows that the best performance is obviously achieved for the local cachable memory scenario, while the worst performance is delivered by the remote uncachable memory. It can also be seen that remote memory benefits significantly from caching, pushing the performance to similar levels like local uncachable memory.

Regarding the RAMDisk scenario, we can see how the performance plummets as soon as the application footprint exceeds the available free memory. In the matrix multiplication and FFT cases we can see an exponential degradation of



(a) Matrix multiplication

(b) FFT

(c) Radix

(d) LU

Figure 4. Application performance depending on the location and type of memory

the execution time, while in the RADIX case this worsening is not that much, probably due to the sequential access pattern and the reutilization of memory pages before swapping. The size of the memory footprint in the LU case is small enough to fit in the free memory, so there are no differences compared to the local cachable scenario.

The average slowdown of all problem sizes between local and remote memory, as the most important metric, is displayed in Figure 5. The slowdown greatly depends on the nature of the applications, in particular on their locality. In the case of cachable memory, the slowdown ranges from 8 for RADIX up to 36 in the case of LU. Nevertheless, the reader should remember that those numbers include both the effect of the higher memory access (almost 2μs using FPGA technology) and the effect of having just one outstanding request.

In order to analyze the potential improvements in the latency of a remote access, we have evaluated an FPGA implementation with an HT200 instead of an HT400 interface (although in both cases the FPGA internal data path runs at the same frequency), so that we can see the future trend when increasing the RMC speed. Additionally, we have also run experiments in loopback mode. In loopback, the target memory controller for a remote read is located in the local node but the read request still uses the local RMC. Figure 6 shows the results from experiments: when switching from HT200 to HT400 (note that the HT link speed only contributes about 30% to the overall speed according to our study of the FPGA design), the execution time decreases about 10-20%. When using loopback, the results show that the distance between the application node and the memory hosts only matters the 20% of the execution time. Thus, the design scales very well with frequency (both for the HT link and for the core logic), and a shift to a high performance ASIC technology should dramatically boost application performance.

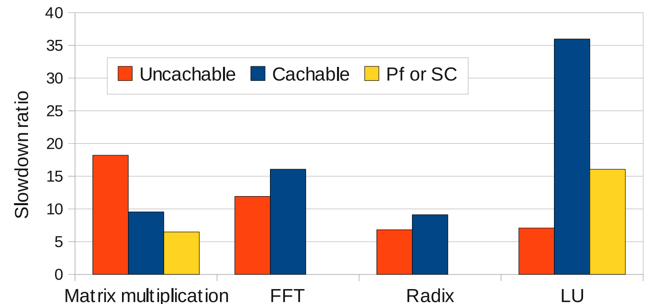


Figure 5. Slowdown as increase of application execution time

B. Multiple Memory Hosts

The previous experiments used remote memory which is only one hop away in the network. Figure 7 compares these execution times to those when leveraging all 32 nodes. In this case, the access latency increases linearly with the

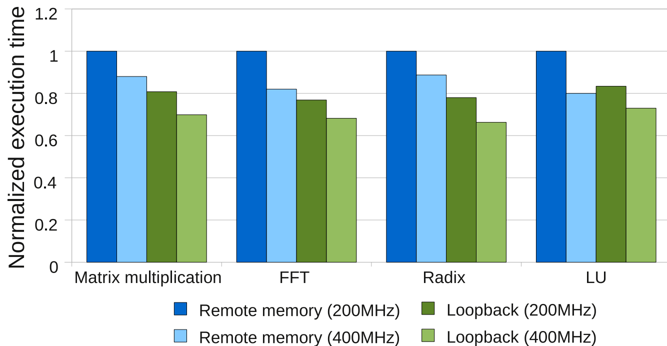


Figure 6. Analysis on the potential FPGA improvement

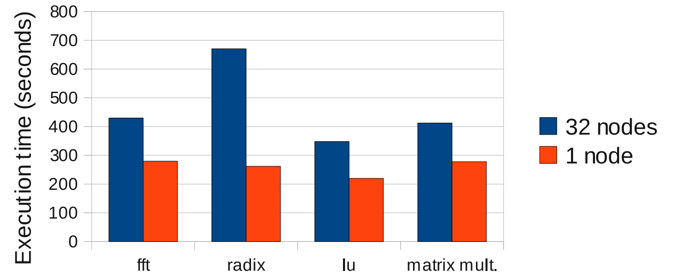


Figure 7. Application execution time when using all 32 nodes

distance in hops between application node and memory hosts. According to the measurements we have carried out, each additional hop adds about 600ns to the memory access latency. Because of this, the execution time increases also as more nodes are involved. It is an indisputable fact that our current topology favors fewer memory hosts, but increasing memory requirements will likely only be satisfied by a large number of memory hosts. Please note that our technology is not dependent on a certain topology.

The right part of Figure 8 also addresses this issue, reporting the increase in execution time depending on the number of hops between application node and memory host. Again, taking into account that our inferior technology results in a hop latency of 600ns, any ASIC implementation should dramatically lower this penalty. We expect a per-hop latency between 50 and 100ns for a well-designed ASIC solution.

C. Memory Host Load

All previous results are derived from unloaded memory hosts, i.e. there is neither load on the CPU nor on the memory. Only the application nodes are loaded by running the application to be characterized. As we propose to use memory from other nodes in a system which will be typically serving for other tasks, we now introduce some load on the memory hosts. In particular, we stress their memory subsystem by running the STREAM benchmark. We also vary the number of threads running this benchmark from 1 to 16, resulting in different memory pressures. Concurrently, an application node uses such a node as memory host by running the matrix multiplication. In the left part of Figure 8 we show the impact of a loaded memory host: for up to 4 threads the impact is almost negligible.

D. Reducing the Slowdown

Results in the previous section show a non-negligible execution time slowdown when using our architecture. There are several reasons for such a slowdown. First, we use FPGAs in order to prototype our proposal instead of ASIC chips. Second, having only one single outstanding request greatly penalizes the performance of our prototype.

Nevertheless, there are other mechanisms that could be included in our design in order to reduce the negative impact of the higher access time. The most prominent one is the use of prefetching. The simplest way to implement such enhancement would be including a linear prefetching in the RMC, so that whenever a new cache line is requested, the RMC retrieves that cache line and the n following ones, although more complex policies could be implemented [18]. These prefetched lines can be stored in a private memory located at the RMC.

In order to easily assess the benefits of using a linear prefetching without having to incur in the large cost of extending the RMC design, we have augmented the matrix multiplication application so that this prefetching is performed by the application itself, and thus the line prefetching and computation for the ongoing matrix multiplication is overlapped. The results of this early implementation can be seen in the data from Figure 4(a) annotated with Pf. For both local and remote memory this mechanism noticeably improves performance. Note that in our current implementation, because the system does not differentiate between prefetch and load transactions, the application still suffers from having only one outstanding request.

Another characteristic that considerably affects application performance is that cache lines brought from remote memory are in the shared state instead of in the exclusive state. The reason for this is the way Opteron processors deal with I/O units (RMC is an I/O unit instead of a regular memory controller). Because an evicted shared cache line from L1 is neither stored in L2 nor in L3 levels (this is imposed by the Opteron design), we cannot leverage these caches for our applications. Moreover, cached lines that correspond to memory mapped I/O behave in a kind of write-through policy: a write operation invalidates the cached line and writes it to remote memory. In this way, a subsequent read to that line will undergo the remote memory latency. By using AMD's CodeAnalyst Performance Analyzer, we checked that LU is particularly suffering from this. In order to bypass these limitations and get closer to the local memory behavior, we have augmented the LU program with

a software caching (SC) technique: prior to a regular access, data is copied from remote memory to local memory and after being used, it is copied back. In spite of the penalty of double copying, we still see a performance increase, which is shown in the data annotated with SC in Figure 4(d).

The performance impact of both techniques (prefetching and software caching) is also clearly visible in Figure 5, reducing the slowdown for the matrix multiplication down to 7x and for the LU down to 16x. Note that a more elaborated design of the RMC, such as a coherent version, would automatically avoid the problem that in this paper we solve by means of software caching, thus increasing performance. Additionally, it would also diminish the need for prefetching as up to 8 outstanding requests would be in fly.

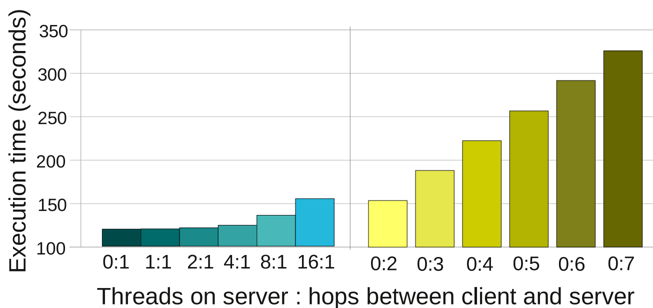


Figure 8. Impact both of load on the memory host (left) and network size (right)

VI. Conclusions

In this paper we have presented a prototype that implements our proposal for memory aggregation across a cluster. Any application can use the entire memory present in the cluster expecting good scalability, as applications do not undergo the coherency protocol overhead that typically limits the size of shared-memory mainframes.

Although our prototype obviously presents worse performance numbers than local memory, we are confident that improved ASIC implementations instead of FPGA, the increase in the number of outstanding requests, and the use of hardware prefetching techniques will bring the performance close to that of local memory. Additionally, note that our system directly allows the use of NUMA policies implemented by current OSes that, once properly configured, may also contribute to reduce the impact of remote memory. In this way, it would be feasible to execute applications with huge memory footprints in a scenario (cluster of computers) where it was not possible before, providing an inexpensive alternative to the costly current solutions.

Acknowledgment

This work was supported by the Spanish MEC and MICINN, as well as European Comission FEDER funds, under Grants CSD2006-00046 and TIN2009-14475-C04.

References

[1] "IBM z Series," http://www.ibm.com/systems/z, IBM.

[2] P. Conway *et al.*, "Blade Computing with the AMD Opteron Processor (Magny-Cours)," *Hot chips 21*, Aug 2009.

[3] S. Kottapalli and J. Baxter, "Nehalem-EX CPU Architecture," *Hot chips 21*, Aug 2009.

[4] "NUMAChip," http://www.numachip.com/, Numascale.

[5] J. Gray *et al.*, "Scientific Data Management in the Coming Decade," *SIGMOD Rec.*, 2005.

[6] H. Montaner *et al.*, "A Practical Way to Extend Shared Memory Support Beyond a Motherboard at Low Cost," in *High Performance Distributed Computing*, June 2010.

[7] J. Oleszkiewicz *et al.*, "Parallel Network RAM: Effectively Utilizing Global Cluster Memory for Large Data-Intensive Parallel Programs," in *International Conference on Parallel Processing*, Aug. 2004.

[8] "Violin Memory," http://violin-memory.com, Violin Memory.

[9] "Altix UV: The World's Fastest Supercomputer," http://www.sgi.com/products/servers/altix/uv/, SGI.

[10] "3leaf Systems," http://www.3leafsystems.com, 3leaf Systems.

[11] M. Chapman and G. Heiser, "vNUMA: a virtual shared-memory multiprocessor," in *USENIX'09: Proceedings of the 2009 conference on USENIX Annual technical conference*. Berkeley, CA, USA: USENIX Association, 2009, pp. 2–2.

[12] "ScaleMP," http://www.scalemp.com, ScaleMP.

[13] M. L. Norman and A. Snavely, "Accelerating data-intensive science with gordon and dash," in *TG '10: Proceedings of the 2010 TeraGrid Conference*. New York, NY, USA: ACM, 2010, pp. 1–7.

[14] T. S, V. C. V, and R. Parthasarathi, "Design-space exploration of flash augmented architectures," *International Conference on High Performance Computing*, 2008. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.164.3227

[15] J. Carter and K. Rajamani, "Designing energy-efficient servers and data centers," *Computer*, vol. 43, no. 7, pp. 76 –78, jul. 2010.

[16] "HyperTransport Technology Consortium. HyperTransport I/O Link Specification Revision 3.10," 2008, available at http://www.hypertransport.org.

[17] H. Fröning *et al.*, "The HTX-Board: A Rapid Prototyping Station," in *3rd annual FPGAworld Conference*, Nov. 2006.

[18] F. Dahlgren, M. Dubois, and P. Stenström, "Sequential hardware prefetching in shared-memory multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, no. 7, pp. 733–746, 1995.