Montaner Más, H.; Silla Jiménez, F.; Fröning, H.; Duato Marín, JF. (2012). A new degree of freedom for memory allocation in clusters. Cluster Computing. 15(2):101-123. doi:10.1007/s10586-010-0150-7.

Noname manuscript No.
(will be inserted by the editor)

A New Degree of Freedom for Memory Allocation in Clusters

Héctor Montaner · Federico Silla · Holger Fröning · José Duato

Received: date / Accepted: date

Abstract Improvements in parallel computing hardware usually involve increments in the number of available resources for a given application such as the number of computing cores and the amount of memory. In the case of shared-memory computers, the increase in computing resources and available memory is usually constrained by the coherency protocol, whose overhead rises with system size, limiting the scalability of the final system. In this paper we propose an efficient and cost-effective way to increase the memory available for a given application by leveraging free memory in other computers in the cluster.

Our proposal is based on the observation that many applications benefit from having more memory resources but do not require more computing cores, thus reducing the requirements for cache coherency and allowing a simpler implementation and better scalability.

Simulation results show that, when additional mechanisms intended to hide remote memory latency are used, execution time of applications that use our proposal is similar to the time required to execute them in a computer populated with enough local memory, thus validating the feasibility of our proposal. We are currently building a prototype that implements our ideas. The first results from real executions in this prototype demonstrate not only that our proposal works but also that it can efficiently execute applications that make use of remote memory resources.

Keywords Cluster · Memory aggregation · HyperTransport

H. Montaner
Universitat Politècnica de València
Departament d'Informàtica de Sistemes i Computadors
E-mail: hmontaner@gap.upv.es

F. Silla
Universitat Politècnica de València
Departament d'Informàtica de Sistemes i Computadors
E-mail: fsilla@disca.upv.es

H. Fröning
University of Heidelberg
Computer Architecture Group
E-mail: froening@uni-hd.de

J. Duato
Universitat Politècnica de València
Departament d'Informàtica de Sistemes i Computadors
E-mail: jduato@disca.upv.es

1 Introduction

High performance computing (HPC) has been traditionally addressed by parallelizing applications into a number of concurrent flows as large as possible and by providing them with the required hardware that supports such level of concurrency. In many cases, that parallel hardware is designed to have as many computing engines as possible as far as its scalability is ensured. The IBM Blue Gene/P supercomputer [25] is a good example of this approach. These large systems are always message-passing platforms, as it is well known that large scale shared-memory systems have never been feasible due to the overhead introduced by the coherency protocol. Nevertheless, it is easier to make the most of a coherent shared-memory computer than of a message-passing one (load balancing, resource sharing, ease of programming, etc). This characteristic is the reason why shared-memory machines are the preferred choice for small to medium computational requirements. Current implementations of this architecture can be found in the IBM z series [4] mainframe, which provides applications with a relatively large number of computing engines and with an amount of memory that can be as large as two Terabytes. Unfortu-

nately, these large shared-memory machines are extremely expensive, and therefore their use is limited to the cases where the features they provide are mandatory for a given computational need.

On the opposite end of the technology market we find computers based on the x86 architecture. They are relatively inexpensive coherent shared-memory machines that are able to scale up to 64 computing cores by using the last developments by AMD [18] or Intel [27]. Additionally, their memory capacity may be as high as a few hundred Gigabytes[1], actually becoming small versions of the expensive mainframes mentioned above. These x86-based machines are currently used as the building block for clusters, which are a cheap and powerful choice for HPC. However, these clusters do not provide a global shared-memory system. On the opposite, they are a composition of small coherency domains, each of them constrained to the boundaries of a given motherboard.

In order to provide a single distributed shared-memory system from a cluster, several solutions have been devised, like the NumaChip by Numascale [7], that provides a coherent distributed shared-memory system across the cluster glueing together all the computing cores and memory resources. The new SGI Altix UV architecture [37] is another example of such resource aggregation. However, these architectures pay the penalty of a lack of scalability and a larger memory access latency due to the limitations and overhead imposed by the protocol that keeps coherency among the nodes of the cluster. Note that this inter-node coherency protocol is different from the one implemented by processor manufacturers in their designs and it could be seen as an additional level of coherency running on top of the intra-node coherency protocol.

Solutions like the NumaChip or the SGI Altix may allow applications to span to as many cores as required and to use as much memory as needed, with the obvious limitation that the amount of required resources must not exceed the amount of resources present in the cluster, which might be quite large. However, many applications may not require such a large amount of computing cores because they may not efficiently scale up to such number of concurrent flows. Actually, the scalability of many applications may be lower than the number of cores located in a single motherboard [3][24] although, they may still benefit from the large amount of memory present in that coherent distributed shared-memory cluster. Therefore, in a cluster executing applications that do not require more cores than available in a single motherboard, there is no real need to provide coherency among processors located in different nodes if every thread from a given application is confined to the processors in the same motherboard. The key reason is that there is no memory sharing among applications, that is, their memory maps do not ever overlap and, therefore, the set of caches that can hold a memory block is restricted to the caches contained in a motherboard. Thus, in this scenario where the memory available to the processors in a given motherboard will probably span to other nodes in the cluster, but all the caches involved in the execution of a given application will remain in the same motherboard, there is no need to propagate coherency operations (e.g. probes and invalidations) to caches out of the node where the coherency operation is started. Nevertheless, coherency among the caches inside a motherboard is still guaranteed by the original protocol implemented by the cores. It can be seen that, in this context, aggregation techniques like the NumaChip or the SGI Altix where every cluster resource (memory and cores) is lumped together 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.

In this paper we propose a practical way to provide non-coherent distributed shared-memory in clusters, thus avoiding the penalty due to the inter-node coherency protocol. In summary, our proposal dynamically partitions the cluster into non-overlapping coherent domains, each of them containing the cores and caches of a single motherboard and perhaps spanning to memory located in other motherboards. In this way, applications that do not scale beyond the number of cores present in a node could still benefit from large amounts of memory by borrowing it from the other nodes of the cluster. Note that in our proposal there is still one independent operating system at each node.

Our proposal has been devised keeping in mind that the final system should be noticeably cheaper than current solutions for large-scale shared-memory and that neither modifications to the application code nor recompiling them should be required. However, this new approach needs two problems to be addressed in order to be efficient. First, the latency of accessing remote memory should be kept as low as possible. Second, as current processors are not designed to access memory with relatively large access time, their performance may noticeably decrease. In this paper we analyze the viability of this new approach regarding these two concerns. Nevertheless, note that other solutions, like the NumaChip or the SGI Altix, are also affected by these two problems.

Before continuing, it is important to emphasize the differences between our proposal and the remote swap

[1] This limitation is imposed by mainboard manufacturers.

technique, that will be deeply discussed later. The remote swap technique is based on page fault interruptions that are handled by the operating system, and this software overhead makes the difference. As we will explain later, our system is not interruption-driven as it does not operate at the page-level, this is, no software is involved. The cornerstone of our system is the fact that we enhance the behavior of the hardware so that there is no conceptual difference between accessing local or remote memory, except for a larger access time.

The remainder of this paper is organized as follows: in the next section there is a discussion about the usefulness of a non-coherent shared-memory system like the one we are proposing. In Section 3 we present a summary of related work. The insights of the proposed architecture are described in Section 4. Section 5 presents simulation results showing an estimation of the performance of our proposal. Section 6 describes the prototype we are currently building to demonstrate this new architecture. Performance results from this prototype are shown in Section 7. In Section 8 some future work is described. Finally, in Section 9 some brief conclusions are presented.

2 On the Usefulness of Non-Coherent Shared Memory Systems

In this paper we propose a novel approach that is only valid for a particular type of applications: those that do not require more cores than available in a single motherboard but may benefit from a large amount of memory. Satisfying only this particular type of applications would usually mean that the usefulness of our proposal is quite limited. However, given the current (and also near to mid-term) trends in processor development, motherboard implementations, and parallel programming, our proposal for non-coherent distributed shared-memory in clusters is very promising. The key for our proposal to succeed is that, on one hand, shared-memory parallel applications do not usually scale beyond a few tens of concurrent flows and, on the other hand, current motherboards can allocate up to 64 cores, while in the future this number may probably increase, making our proposal even more appealing because it will satisfy a larger number of applications.

The system we are proposing provides several additional advantages. It not only will allow to run applications that require large amounts of memory at low cost, but it will also reduce the cost of acquisition of clusters by reducing the amount of RAM installed in the nodes of the cluster. Effectively, nowadays the memory capacity at each node is overscaled, just in case a process may need a big amount of memory sometime. The rest of the time that memory remains unused [11]. By leveraging our proposal, memory resources at each of the nodes do not need to be oversized, because in case an application (parallel or sequential) requires more memory than the physically available at that node, it can borrow additional memory from other nodes. Nevertheless, note that the objective of our proposal is not making the most of idle memory in a cluster. We simply propose a novel way to allow an application that presents a large memory footprint to be efficiently executed with a low-cost infrastructure. Therefore, in case there is no free memory available in the cluster and an application requires some additional memory, our proposal is still valid. In this case, the cluster administrator would be responsible for freeing memory somewhere in the cluster according to the priority of applications.

Another area where our proposal could be useful is data centers where we can find a heterogeneous range of memory-hungry applications. On one hand, there are some applications that can be sped up by providing more memory, for example in-memory databases [23][5] and datamining [39]. The idea behind these databases is storing some or all of the tables of the database in main memory. In this way, the access to those tables would be much faster than when using traditional harddrives. It would also be much faster than using SSDs (solid-state disks) for storing the database. However, in order to leverage this approach for large databases, the database server should be configured with hundreds of GB of memory (or even several TB). This amount of memory is not feasible in a single mainstream x86-based server.

Other memory-hungry applications found in data centers are those whose execution is prohibitive without enough available memory, like some kind of simulations [3], scientific applications [24], etc. For example, in the chemistry domain, the gaussian application requires huge amounts of memory, despite it only scales up to a few parallel threads.

A third memory consumption paradigm is virtualization: servers are often partitioned so that they can simultaneously execute several operating systems (OSes) and thus provide service to several customers in such a way that they believe that they own an entire computer. Moreover, virtualization usually achieves a higher productivity rate (number of used cores). Typically, the memory granted to a given virtualized OS is not always completely in use; this fact makes possible to execute these virtualized OSes in a computer with a physical memory size smaller than the sum of the memories seen by each OS. But if several of the virtualized OSes require all their granted memory at the same time, then they would outgrow the node's physical memory. In this

case it would be helpful to provide additional memory resources from other nodes in order to avoid swapping to disk, what would noticeably slow down the virtualized machines. Note that as coherency is only required inside each of the virtualized OSes but not among them, memory borrowed from a remote node and used by a given virtual OS will not require to be coherent with the memory used by a virtual OS executing in that remote node. This is just the model our proposal is intended for.

Although the primary idea of our proposal is to provide a cost-effective way of extending the memory used by a process to remote memory located in other nodes of the cluster, it could also be used for communication among processes running at different nodes. Effectively, our proposal can be easily adapted to a communication paradigm like PGAS (*Partitioned Global Address Space*) [15]. The performance of this programming model can equal that of MPI codes and, for most humans, it is much easier to learn [41]. Also, PGAS is not less scalable than MPI and permits sharing, whereas MPI rules it out [42]. On the other hand, PGAS implements a one-sided communication model (faster than two-sided), where caching is not required and the programmer makes local copies and manages their consistency. Therefore, no cache coherence protocol is needed, except between the network interface and the processes in a node. Thus, PGAS perfectly matches the system architecture we are proposing in this paper.

As can be seen, although the scope and usefulness of our proposal seemed to be quite limited at the beginning, there are many different application domains that could take advantage of it.

3 Related Work

Disk swapping is the traditional approach for executing applications with a memory footprint that exceeds the available physical memory. This technique turns the hard disk into another level of the memory hierarchy so that not recently used data is moved up to disk to free space in main memory. However, when the working set of an application is bigger than the available main memory, the thrashing problem easily arises. Once this state is reached, execution time increases to a prohibitive level and system performance plummets.

The most extended technique in the academic scene for getting additional memory is remote swap [28][34][35]. This technique moves pages from main memory in the local computer to memory in other computers of the cluster, aiming that retrieving those remote pages will be faster than retrieving them from hard disk. Previous studies have proved that, even on a regular Ethernet network, a remote memory access made across the network is slightly faster than a local disk access [12]. However, remote swap presents the same drawback as traditional swap, which is not only affected by disk latency, but it also suffers from software latency, that is, the time required by the operating system to swap memory pages. Thus, although remote swap is free from disk latency, it also suffers from the operating system overhead and therefore, it does not overcome the thrashing problem.

A different approach is followed by Violin Memories, that offers a memory server that can hold up to 504 GB of RAM [9]. This server is attached to the computer by means of a PCI-E adapter. Unfortunately, this solution not only lacks from scalability (no more memory than 504 GB) but it also presents a large access time (3 microseconds) because the OS is required in order to access that extra memory. Additionally, it is an expensive approach (a server populated with only 120 GB costs more than $20.000).

Other companies are working on providing more resources to applications from a different perspective, which is based on aggregating all the resources in a cluster into a single computer. 3Leaf [1], ScaleMP [8], Numascale [7], and SGI [37] are examples of these companies that not only provide global access to the cluster memory, but also to the processors and other resources.

In the case of ScaleMP, a virtualizing software layer is provided so that multiple x86 systems are aggregated into a single virtual x86 system, delivering a virtual symmetric multiprocessor system where coherency is maintained among the integrating x86 computers. Although this approach provides a chip solution for coherent shared memory, its main drawback is that it is software based, thus reducing performance. Actually, a single memory reference going through the vSMP ScaleMP layer takes 25 microseconds. Another example of software aggregation is vNUMA [14], although it presents similar latency problems.

3Leaf, Numascale, and SGI follow a similar approach but from a hardware-based perspective. 3Leaf provides systems leveraging their Aqua chip, which provides a coherent shared-memory system that is the aggregation of multiple x86 systems. Numascale makes leverages the NumaChip to provide similar features. SGI makes use of its proprietary UV_HUB in order to provide resource aggregation. Nevertheless, as they provide access to processors in other motherboards, coherency must be maintained throughout a large number of computers, limiting the scalability and performance of these proposals in practice. This is, actually, the main difference between these approaches and our proposal, where coherency among nodes is not kept.

A different proposal from industry is IBM's Dynamic Logical Partition (DLPAR) [2], which reassigns memory inside a coherency domain. Basically, it moves memory from one process to another, both of them being executed in the same coherent shared-memory computer. Our proposal is quite different because it borrows memory from a coherency domain and logically moves it to a different coherency domain.

Finally, other studies show the interest in aggregating somehow the memory in a cluster [20]. However, these studies differ from our system in that their implementations operate at page level, and this implies the use of software layers (OS kernel or hypervisors), while in our proposal no software is involved in accessing remote memory, as it relies only on hardware.

3.1 The SGI Altix UV Architecture

As a recent commercial example of resource aggregation in clusters providing a single global shared memory system, in this section we review the SGI Altix UV architecture.

The SGI Altix UV is the fifth generation of SGI's scalable global shared memory architecture, which scales up to 2048 cores and up to 16 TB of memory. These systems are built using the SGI NUMAlink interconnect, that provides the high-bandwidth and low-latency required by these global shared-memory systems.

The building block of the SGI Altix UV system is a compute blade containing one or two processor sockets, each of them capable of supporting either 4-, 6-, or 8-core Nehalem EX processors. Each socket is attached four DDR3 memory channels and four Intel QuickPath Interconenct (QPI) connections, that allow processors to communicate with each other and with the UV_HUB ASIC developed by SGI, whose main purpose is glueing all the different computing blades into a single coherent system. As can be seen, a SGI Altix system is basically a cluster of standard Nehalem-based nodes interconnected by means of an ASIC chip designed by SGI and incorporated into each of the motherboards.

The two initial products of the family are the Altix UV 100 and the Altix UV 1000. The Altix UV 100 is aimed at the mid-range market, scaling from a single unit containing two dual-socket blades up to a 96-socket machine that fits into a couple of racks. The maximum memory capacity in this product is 6 TB, while it provides a maximum 96-socket configuration (768 cores are available). The Altix UV 1000 is a cabinet solution that scales up to 256 sockets, yielding 2048 cores and 16 TB of memory. Both the Altix UV 100 and the Altix UV 1000 provide a single system image, that may be leveraged by means of either SUSE Linux Enterprise Server or Red Hat Enterprise Linux. However, the Altix UV 1000 system also allows larger scalability if the global system is partitioned into several independent system images. The maximum size is 16384 nodes with 32768 sockets and 262144 cores. In this configuration, the maximum amount of memory is 8 PB.

One of the key components of an Altix system is the UV_HUB chip, an ASIC developed at SGI that links the cache-coherent Intel QPI interconnect with the larger cache-coherent NUMAlink environment that extends across the full Altix UV system. The NUMAlink interconnect is proprietary from SGI. The UV_HUB chip additionally provides efficient support for some MPI operations.

The Altix UV platform can be interconnected using different topologies depending on system size. The smallest platform size is interconnected by a switched dual-plane topology with a maximum of three NUMAlink hops between any node. Medium-size systems with up to 16 blade chassis in 8 racks (512 blades, 1024 sockets, 8192 core) are interconnected in a fat-tree topology using the required number of 16-port NUMAlink routers. Larger configurations are achieved using 256-socket fat-tree groups connected in a 2D torus. As can be seen, the NUMAlink interconnect developed by SGI provides external routers where the UV_HUB chips are connected to. In any case, remote memory access latency is lower than 1 microsecond inside a single system image. It increases up to 2 microseconds for traversing the largest configuration.

Unfortunately, we cannot make a price comparison of the SGI Altix UV platform because SGI provides prices only under NDA.

4 A New Shared-Memory Architecture

Our aim is to provide additional memory to processes requiring it by logically assigning them memory that is physically attached to other computers in the cluster. As mentioned before, it is common to reach a situation where processes in a node require more memory than available in that node. In this case, memory from other nodes may be used to expand the available memory resources of those processes at almost no additional economic cost. It is important to remark that there is one independent operating system at each node, and that a process is confined to the processors and caches located in the node where it is being executed. However, the system we are proposing breaks the inter-node border and allows a process to dynamically use memory initially owned by other operating systems.

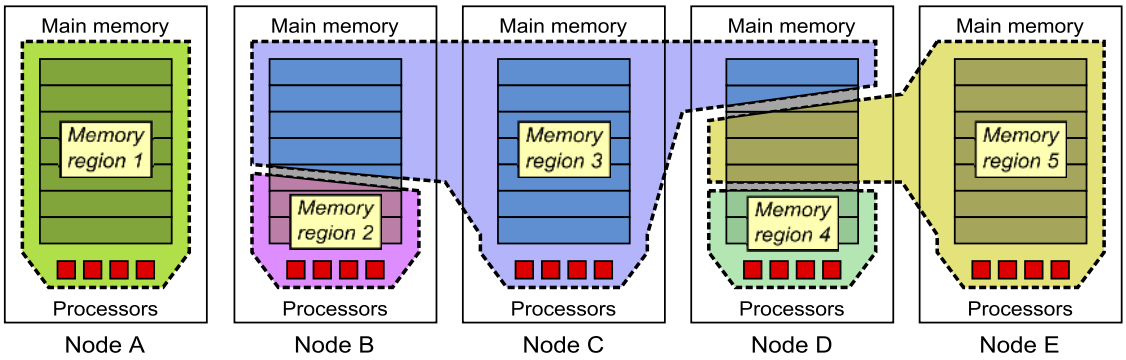


Fig. 1 An example of memory sharing among the nodes of a cluster

In this section we present in detail the key component of our proposal: how to efficiently access memory located at other motherboards. Note that deploying the full system we are proposing requires additional components, not described in this paper due to space limitations, such as:

- modifications in the OS in order to augment the system calls that reserve and dispose memory, so that they can do so with remote memory,
- augmenting the OS services so that knowledge of the location of free memory across the cluster is achieved,
- a communication protocol among nodes suitable for reserving and disposing remote memory and also for accessing that memory,
- concerns related to communication reliability and security,
- a network fabric that interconnects all the nodes with low latency and high throughput,
- other OS related topics like the use of remote memory as OS buffer cache, memory migration, fragmentation prevention, etc.

4.1 System Overview

To understand what our system does (and what it does not), 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 five nodes of a cluster and five memory regions. Region number 1 is confined to node A and represents the default configuration for a node, that is, processes in that node can access the entire node's memory. On the other hand, region number 3 has been extended to the neighbors of node C, so processes in this node now have 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. Finally, region 5 has been also extended to its neighbor node D, where three memory regions coexist. Moreover, although enlarged memory regions in Figure 1 have spanned to their neighbor nodes, this is not a requirement in our system. Actually, a node may extend its memory resources by borrowing memory from any node in the cluster. Finally, note that in our proposal there will be as many memory regions as nodes in the cluster because processors in a given node will always create a memory region, independently from processors in the other nodes. What can be dynamically adjusted is the amount of memory for a given region.

It is important to emphasize that as memory regions are independent, processes in node A can only access region 1, processes in node B can only access region 2, processes in node C can only access region 3, etc. In the same way, as all processors in a node can only access one (the same) memory region, all caches in a node will only cache data from one (the same) memory region. This is the reason for the good scalability of our proposal. 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 that region is limited to the caches in a node. In other words, as each memory region is an independent coherency domain, a processor bound to a certain memory region does not need to know what happens in other regions, and thus changes in a memory region are only notified to the caches of that memory region. No matter how large the region is, only the caches contained in one node will be informed. As can be seen, our system decouples memory from processors, and there-

fore there is no coherency overhead when aggregating huge amounts of memory.

Finally, our system does not rely on any kind of run time or communication library. The core of our system is a quite simple piece of hardware, as will be shown next. Additionally, the process of accessing remote memory completely relies on hardware and is therefore free of any software overhead. This is a key feature over other solutions where a software layer penalizes every access to remote memory. In our proposal, a regular load or store operation issued by an application will trigger the hardware mechanism to access data from remote memory. Because of this characteristic, the time required by a remote access can be very low. The way we accomplish this is by means of HyperTransport.

4.2 System Architecture

HyperTransport technology [10] is currently the lowest latency, highest bandwidth openly licensed standard communication technology for chip-to-chip and board-to-board interconnects. We can find its flagship implementation inside the AMD Opteron processor [26], where HyperTransport is used to interconnect the processors in a motherboard. In these systems, each processor is attached to part of the physical memory by means of its own memory controller, as shown in Figure 2(a). Therefore, as there are several memory controllers in the system to access memory, processors require to know where to forward a given memory request. This is achieved by including at each processor a set of base and address registers (BAR) configured at the initialization phase that reflect the system physical memory distribution. In this way, 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 operation to the memory controller responsible for that memory address, provided by the previous comparison. Forwarding the memory operation involves the generation of a HyperTransport message.

The system described above is the basis upon which we will design the technology that enables the access to remote memory. Our proposal involves creating a new hardware component that will implement the required functionality. Hereafter, we will refer to this component as *Remote Memory Controller* (RMC). This new component will be presented to the processors in the motherboard as a new memory controller as shown in Figure 2(b). However, the RMC will not be a regular memory controller as it has no memory banks directly connected to it, otherwise it relies on the memory banks installed in other nodes in the cluster. In order to enable the

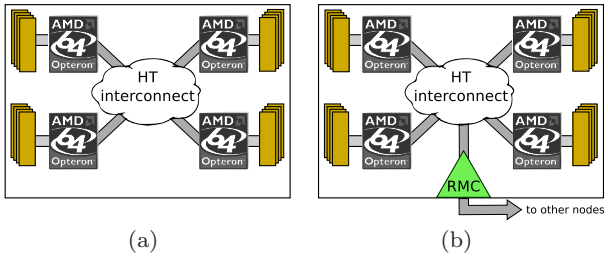


Fig. 2 (a) Motherboard diagram showing four processors interconnected by means of HyperTransport. (b) A Remote Memory Controller has been attached to the motherboard.

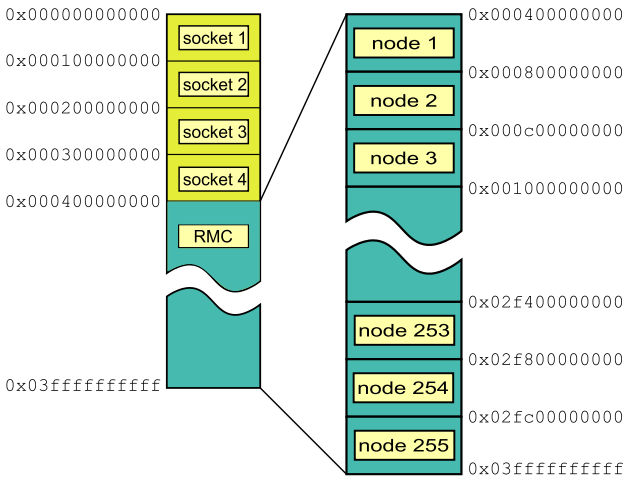


Fig. 3 Example of the memory map of a node in a 255-node cluster. Node identifiers range from 1 to 255

RMC functionality, the BAR registers mentioned before must be reconfigured so that some of the memory accesses are forwarded to the RMC, that will convert those accesses into remote accesses. Note that this system model does not imply modifications in the node architecture. On the contrary, it only requires a new card containing the RMC to be added to the existing nodes, as we will explain later.

Figure 3 is a representation of the shared memory distribution seen by a node in a 255-node example cluster (having 255 nodes instead of 256 will simplify the RMC design, as it will be shown later). In this example, each node has 4 sockets, each of them attached to 4 GB of main memory. Nevertheless, the node can see a memory space of 4 TB of main memory. This is because above 16 GB, the memory is mapped to the RMC, as shown in the right column. As can been seen, the 14 most significant bits of the memory address determine whether a memory operation must be forwarded to a local memory controller or to the RMC[2]. If those 14 bits are all set to zero, then some local controller will own

[2] Depending on the amount of nodes in the cluster and on the memory each node has, the number of most significant bits that determine whether a memory location is local or remote will change.

the required address. Otherwise, the RMC will manage the operation by forwarding the memory request to the corresponding node pointed by the 14 most significant bits. When the memory operation arrives at the destination RMC, that RMC sets to zero those 14 bits and forwards the operation to its local system by generating the appropriate HyperTransport message. Once the RMC in the remote node gets the response message from a memory controller in its motherboard, it forwards the response to the source RMC. As can be seen, the Opterons in the local motherboard are not aware about the remote memory and they see the RMC as another memory controller (with a huge quantity of memory behind it).

There is an important detail that requires further explanation. As pointed out before, in order to simplify the design of the RMC, our example cluster is composed of 255 nodes. Node identifiers start at node number 1 and finish at node number 255. Our system will never have a node identified as node 0. By doing so, every node has an identical physical memory map conception, that is, local memory at each node always starts at address `0x000000000000`, as it is represented in the left side of Figure 3. On the other hand, remote memory will always be denoted by the 14 most significant bits being different from 0. In this way, programming the set of registers used to forward memory accesses is simplified as well as the design of the RMC, which will not require any kind of translation table. Unfortunately, there will be an overlapped segment in the memory map for each node. For example, if node 2 addresses memory between `0x000800000000` and `0x000bfffffffff`, it will be referring to its own local memory. However, this will never happen thanks to the way memory is reserved, as explained next.

It has been described in the previous paragraphs how a processor automatically forwards memory accesses to a remote node. Nevertheless, before accessing memory (local or remote) it is always necessary to reserve it for the process that will make use of it. The way memory is reserved in our system is crucial because if that reservation is properly done, then following accesses can be very fast. Software layers are involved in the reservation process, contrary to the process of accessing remote memory where only hardware mechanisms are used. Thus, although the reservation process is not time-critical, it should pave the way for future load and store operations. Next we will expose the reservation mechanism in a naive way for better understanding. Some secondary aspects have been ignored in order to focus on the main process. Additionally, before presenting the reservation mechanism, we should review the basics of virtual memory.

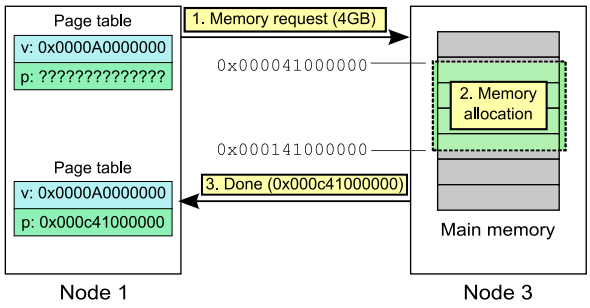


Fig. 4 Node 1 reserves remote memory in node 3

When a current processor issues a load or store operation to access some data in a given memory address, that address is, likely, a virtual address. To carry on the operation, that virtual address has to be translated into a physical one, that is, an address directly referring to a location in a certain memory bank. This is automatically done by the processor by looking up at the Translation Lookaside Buffer (TLB). If the virtual page containing the virtual address has a corresponding translation in the TLB, the load or store operation continues immediately. If the processor does not find a valid entry for that page in the TLB, then it raises a page fault exception that is caught by the operating system. The OS looks up the page table and writes down the translation into the TLB, so that the operation can continue. In case there is not an appropriate mapping in the page table, the OS has to allocate space in the physical memory to the requested page and write down the address in the page table before updating the TLB. In any case, after the translation the load or store operation will be directed to whatever address the operating system wrote into the TLB. This fact, together with the forwarding process based on the memory distribution mentioned before, makes possible that load and store operations can access remote memory in a fast and simple way.

Once the basics of virtual memory have been reviewed, let us introduce the remote memory reservation mechanism, which is carried out by the OSes without any interaction with the RMC. Nevertheless, it will be necessary to add some functionalities to the OS to manage the page table, as shown in Figure 4. This figure presents an example of remote memory reservation. A node in the cluster, for instance node number 1, has a virtual memory area that requires to be mapped into a physical one but does not have a physical address yet. Let us assume that the OS realizes that it is running out of local memory and therefore node 1 needs more memory. Then, somehow it discovers that node 3 has some idle memory available and a message is sent to node 3, asking for some memory to be reserved. After arrival

of the request message, node 3 reserves the requested amount of memory. Unlike the traditional reservation process where physical memory is only assigned when a memory page is accessed, this reservation process actually reserves a zone in the remote physical memory. Let us assume that the reservation is done over a contiguous physical memory area, for example, in the memory area that starts at `0x000041000000` and finishes at `0x000141000000`, this is, 4 GB. The starting physical address is sent back to the requester node in an acknowledgment message. However, one modification is done to that physical address before sending it back: the 14 most significant bits are changed to reflect the identifier of node 3 (note that in a local system those 14 bits are always zero). When node 1 receives the response message, it writes down the translation from virtual to physical memory in the page table. The prefix added by node 3 will be used by the load and store operations to address node 3.

From this point, remote memory accesses will be automatically performed by hardware. After reserving remote memory, a processor in node 1 may issue a memory operation related to virtual address `0x0000A0000B00`, for example. As usual, the CPU will translate this virtual address into a physical one. As the operating system has previously written the corresponding translation into the page table, now the TLB can be immediately updated and the memory operation goes on now with the corresponding physical address: `0x000C41000B00`. The CPU knows that this address is managed by the RMC (Figure 3) and therefore the memory access is forwarded to the RMC, which examines the 14 most significant bits and sends the memory access request to node number 3. When the request arrives at node 3, the RMC in that node will set those 14 bits to zero and transmit the operation to its local system with physical address `0x000041000B00`. In case of a read access, then a response containing data will be sent back to node 1.

As can be seen, this mechanism does not need any kind of translation table in the RMC (thanks to the fact that there is no node 0, as explained before). This allows that very little functionality has to be implemented in the RMC, and thus small overhead due to message processing is generated. Once the remote memory is reserved, that memory will never be accessed by processes being executed in the remote node because the remote OS will never assign that memory to them because it is already reserved. Therefore, there is no need for keeping coherency between the caches in the remote node and the caches in the node that is using that remote memory, as explained before.

4.3 System Implementation

AMD uses HyperTransport to interconnect the different memory controllers in a motherboard. As the RMC is presented to the rest of the processors as an additional memory controller, its design requires providing it with a HyperTransport interface so that it can communicate with the rest of the devices in the motherboard by exchanging HyperTransport messages.

Additionally, the memory accesses that the RMC will receive from the local processors will be forwarded to remote nodes and responses to those remote memory accesses will be received from other nodes and forwarded to the local processors. The RMC will also receive memory accesses requests from other RMCs in the cluster. Thus, in order to allow communication between RMCs, the natural way would be to leverage HyperTransport for inter-node communication. However, this protocol is not able to address more than 32 devices, which would be the general case when deploying our proposal in a cluster. Therefore, for communication among nodes, we will make use of the High Node Count HyperTransport Specification 1.0 recently published [19], which extends HyperTransport's addressing capabilities to address a much higher number of devices. In this way, the RMC will have a regular HyperTransport interface to the local node and a High Node Count HyperTransport interface to the rest of the cluster, bridging from one standard to another. The reader could refer to Section 7.2 in [19] to know how to perform the translation between both standards.

On the other hand, the circuit that implements the RMC needs to be physically connected to the motherboard of the nodes in the cluster. In order to do so two options are feasible. The first one is using an ASIC bridge chip included in the chipset of the motherboard in a very similar way to the bridge implementation proposed in Section 7.2 of [19] to implement the High Node Count HyperTransport specification. The main difference with that proposal is that the new chip should be augmented with the RMC functionality. As this option requires motherboard manufacturers to make a decision for this new technology and this is quite unlikely nowadays, the second option is to make use of HTX compatible cards, able to directly connect to the HyperTransport link. These cards would include the same functionality as in the previous option.

It is worth to mention that as the system size increases (more elements are involved) it is more exposed to failure. In order to deal with these failures, RMCs should track pending transactions, so that a non-satisfied remote memory operation could trigger an interruption in the local system after a given timeout. This interrup-

tion should be caught by the OS and properly handled (typically by killing the affected process).

5 Feasibility Analysis

In this section we analyze the feasibility of our proposal. The characteristic that could most negatively influence the performance of this system is the larger latency of accessing remote memory. We will study its impact on performance by comparing it to the best and worst alternatives. Some refinements intended to hide remote memory access time are also evaluated by simulation.

5.1 Methodology

Here we present a simulated system as a first step towards the implementation of the proposed architecture. Prior to embarking on the hardware particular implementation, we aim in this paper to check the feasibility of our idea and test different configurations in a quick and flexible environment, this is, simulation. In order to analyze how applications are affected by remote memory access latency, a node of the cluster that uses remote memory has been modeled with SIMICS [31]. Moreover, GEMS [32] has been used to model an accurate memory hierarchy. In order to model a system as similar to the Opteron system as possible, the AMD Hammer coherency protocol has been used. The simulated system configuration is a 4-socket motherboard with 256 MB of local memory and 256 MB of remote memory. At this point, the exact location in the cluster of these 256 MB of remote memory is not modeled. We assume that access time to remote memory will be the same regardless of the exact node that owns it. Note that in a real cluster, the total amount of available memory would be much larger than 512 MB. However, the whole system has been scaled down to achieve a reasonable simulation time. On the other hand, Open Solaris 10 has been loaded into SIMICS. We have observed that a cooled down Solaris has a 130 MB resident memory footprint, leaving 126 MB of local memory for applications. Finally, notice that the operating system is not aware of the different memory latencies, so no NUMA support is in use. Note that the lack of NUMA policies will allow us to analyze the lower bound of our system performance. When NUMA is in use, better performance numbers would be achieved.

The latencies used in our model [17][30] are shown in Table 1. A 1-hop memory access takes place when a processor accesses a memory position in the memory banks directly attached to it. 2-hop memory accesses occur when a processor accesses a memory bank attached to another processor in the motherboard. The hit or miss remote memory latencies refer to a private memory cache attached to the RMC as we will explain later. If the requested data is present in the RMC private cache, then no data has to be fetched from a remote node. This cache memory located in the RMC is not part of the memory map of the system and is independent of the coherency system of the processors in the motherboard. It is intended for internal RMC use only as a private cache meant to store last accessed cache lines. On the other hand, latencies in Table 1 labeled as ASIC refer to an implementation of our system in an ASIC chip attached to the motherboard as part of the chipset. Finally, latencies labeled as FPGA denote the latencies achieved by the HTX card containing the FPGA mentioned before.

Parameter	*Latency*
L1 cache	1
L2 cache	6
Local memory latency (1 hop)	100
Local memory latency (2 hops)	140
Remote memory latency (FPGA) hit	450
Remote memory latency (FPGA) miss	2720
Remote memory latency (ASIC) hit	170
Remote memory latency (ASIC) miss	1520

Table 1 Latencies used in the modeled system (cycles in a 2GHz clock system)

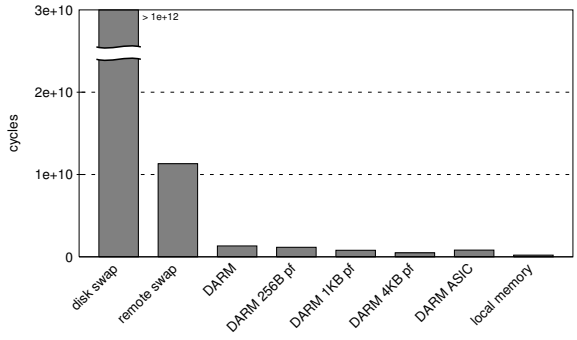
SIMICS also allows to model hard disks. According to main vendors [6], we have modeled a high performance disk with an access time of 3.2 milliseconds and a data transfer rate of 320 MB/s. This will be useful to simulate the traditional disk swap technique. Note that SIMICS is a full system simulator, and therefore it will allow us to take into account the overhead due to the operating system. This overhead will also be considered in the case for remote swap.

Furthermore, we have slightly modified GEMS in order to study the effects of an out-of-order processor that implements the Miss Handling Architecture (MHA). The Miss Handling Architecture [40] is a hardware structure coupled to the processor cache in order to exploit the memory parallelism. A MHA consists of a set of Miss Status Holding Registers (MSHR, also known as TBE) and the number of MSHRs determines the amount of outstanding misses a cache can experience before it blocks. Thus, the more MSHRs a cache has the more a processor can go ahead before stalling. If a cache has few MSHRs, then the processor will probably stall just because no more memory operations can

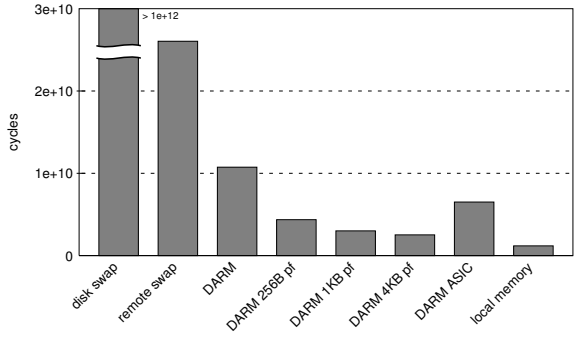
be issued by the cache before responses for outstanding requests arrive. If the cache has many MSHRs, it will not be a bottleneck and the processor will be able to make progress as long as it could find more non-dependent memory instructions. Nowadays, processors implement a number of MSHRs that matches current memory latencies. Therefore, if memory latency is increased, the amount of MSHRs needs to be also incremented in order to avoid losing performance. Current Opterons, for example, implement 8 MSHRs [17].

The reason for the slight modification we have introduced in GEMS is that simulations take a prohibitive execution time when using an accurate out-of-order simulator. Our simplified out-of-order processor does not take into account the dependencies between memory operations, so it is able to issue a new memory operation even if the previous dependent operation has not finished. This introduces some inaccuracy in the simulation results. However, this relaxed model perfectly fits the objective of our study because the only bottleneck in the instruction flow we are analyzing will be the memory. Additionally, our simplified out-of-order processor makes possible the study of the impact of large MHAs.

Finally, regarding the applications to be used in our simulations, we have chosen the STREAM [33] and PARSEC [13] benchmarks. Additionally, a modified version of STREAM that performs random accesses to the vectors will also be used. Note that the regular STREAM benchmark presents a very high degree of locality, while the random STREAM presents a very low amount of locality. PARSEC applications would be in between both STREAM ends.



(a) out-of-order, 8 MSHRs



(b) in-order, 8 MSHRs

Fig. 6 Execution time of random-flavoured STREAM benchmark

5.2 Results

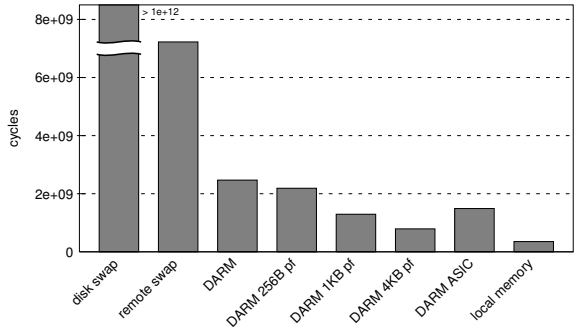
In this section we present the results for the simulated system. First, we carry out a comparison between the performance achieved by our proposal and the performance achieved by a system with enough local memory on one hand, and a system that makes use of remote swap on the other hand. Later, we will study the impact of the Miss Handling Architecture in our non-uniform memory access context.

5.2.1 Remote Swap vs Direct Access

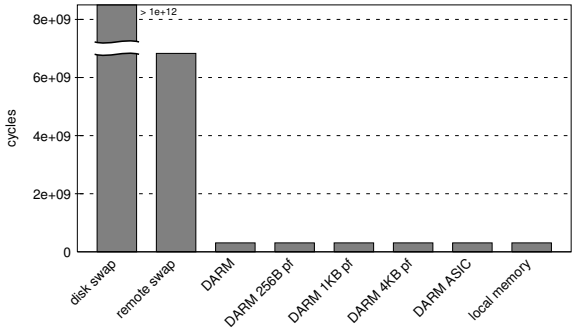
As a first step, our aim is to analyze whether our direct access technique to remote memory is better than remote swap. In order to accomplish this, two scenarios have been modeled: one for the direct access mechanism and one for remote swapping. Both scenarios have 256 MB of local memory and 256 MB of remote memory which is used in two different ways: for the remote swap scenario the remote memory is used for page storing, and for our system the remote memory is used just like the local memory (but with higher access latency). Moreover, two additional scenarios have been evaluated to draw the upper and lower performance limits: traditional disk swap (256 MB of local memory and no remote memory) and local memory (512 MB of local memory and no remote memory).

To model our system, latencies in Table 1 have been used. In order to model the remote swap technique in an optimistic way, retrieving 4 KB pages from remote memory has the same latency as a remote memory access (2720 cycles). Note that in this latency it is not included the software overhead which will be automatically added by SIMICS when dealing with the OS code that carries out the swap operation.

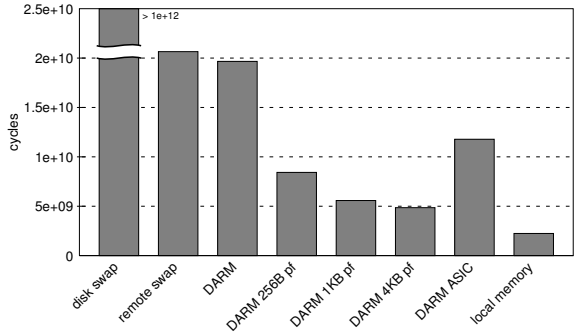
Figure 5 presents the results for the execution of the STREAM benchmark for several case studies. Let us focus on the bars labeled “disk swap”, “remote swap”, “DARM”, and “local memory”. DARM stands for Direct Access to Remote Memory, that is, our proposal, when it is implemented in an FPGA. As can be seen, disk swap is inadvisable in any case, as it takes a prohibitive time to finish the execution. In the other hand,



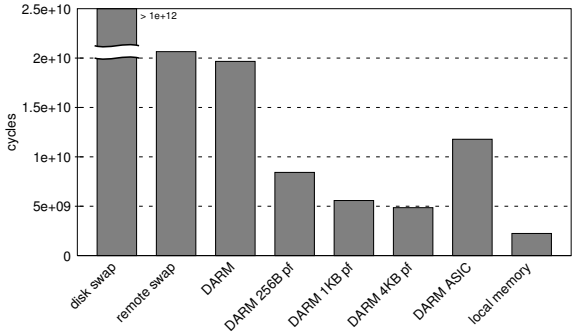
(a) out-of-order, 8 MSHRs



(b) out-of-order, 32 MSHRs



(c) in-order, 8 MSHRs



(d) in-order, 32 MSHRs

Fig. 5 Execution time of STREAM in multiple scenarios

the 512 MB local memory scenario makes possible a very fast execution. By comparing Figures 5(a) and 5(b) we can see the effect of our simplified out-of-order processor, because an increase in the number of MSHRs reduces the execution time of the benchmark. On the contrary, comparing Figures 5(c) and 5(d) we can see that the number of MSHRs has no impact in an in-order processor, as expected.

Regarding our main comparison, the difference between remote swap and DARM varies from Figure 5(a) and 5(c) due to the out-of-order processor. The out-of-order processor accelerates the DARM execution in a factor of 8, but it only accelerates the remote swap execution by a factor of 3. This means that our system is 3 times faster than the remote swap technique in an out-of-order context, and only slightly faster in an in-order context. The reason why the out-of-order processor has little impact in the remote swap execution is that the swap latency cannot be hidden by the MHA, because the page fault handler has to finish, this is, the data has to be retrieved from hard disk, before resuming the execution flow. In the case of DARM, there is no handlers involved, so an access to remote memory does not break the continuity of the execution flow, but it only has higher latency that can be disguised by the MHA. This fact, coupled with the software overhead itself (we have measured an overhead of 2 ms per page fault), makes our proposal an alternative much faster than remote swap.

The main disadvantage of our system is that there is no memory level between the processor cache and the remote memory. This implies that data is fetched with very fine granularity (cache line), while the remote swap technique fetches an entire memory page. One solution to this issue is to provide the RMC with a private memory so that more than one cache line could be retrieved at a time from remote memory. In this way, the RMC would have required data by the processors closer to them in advance. Figure 5 shows results for a linear prefetching improvement for three prefetching sizes: 256 B, 1 KB and 4 KB, that is, every time a cache line is accessed, 256 contiguous bytes (or 1 KB or 4 KB) are retrieved and stored in the private RMC cache. The size of the RMC cache is four times the size of the prefetching in each case. Simulations results in Figure 5 show that, in the case of the STREAM benchmark, the prefetching technique works really well, getting performance quite closer to that of the local memory. The reason why Figure 5(b) presents such flat results is explained in the next section.

Implementing the RMC in an ASIC instead of in an FPGA will reduce the latency of accesses to remote memory. Simulation numbers predict that the performance of our system has the chance to be close to

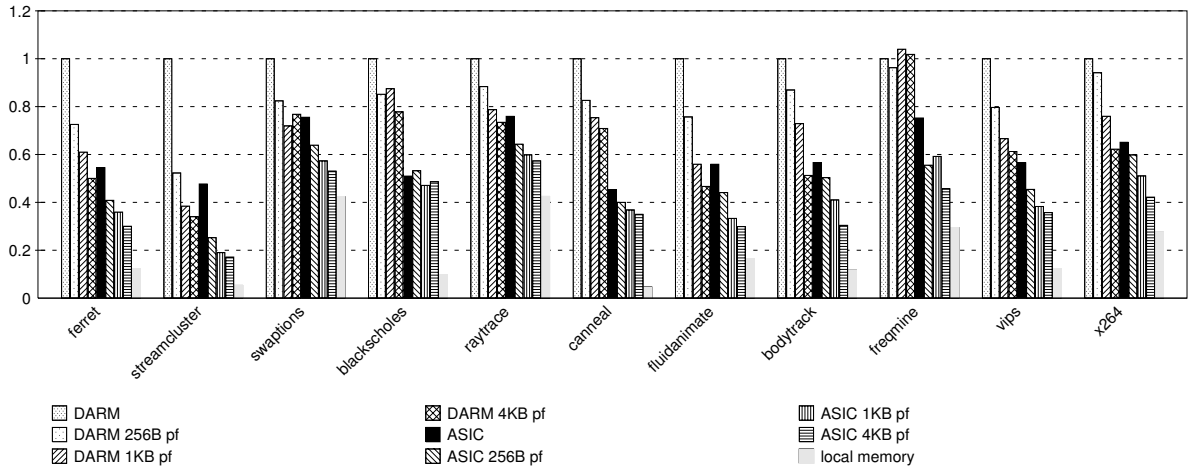


Fig. 7 Normalized execution time of the PARSEC benchmarks

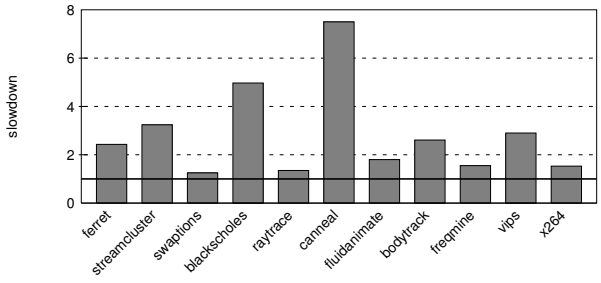


Fig. 8 Undergone slowdown when using remote memory instead of local memory

the performance achieved by a system populated with enough local memory for running the application. Note that no prefetching has been used in the ASIC simulations. Leveraging some kind of prefetching would improve performance as in the case for the FPGA.

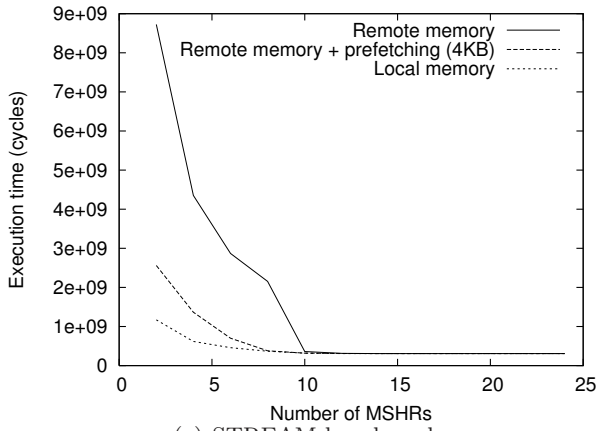
Figure 6 presents the simulation results for the random access benchmark. In this case, memory accesses present much lower locality. Therefore, the remote swap technique needs to swap more pages and the execution time increases in a higher proportion when compared to the DARM technique.

Finally, Figure 7 presents some initial results for the PARSEC benchmarks in an out-of-order context with 8 MSHRs. It can be seen that the prefetching technique comes in useful to the majority of the benchmarks. More sophisticated prefetching policies may reduce even more the execution time by capturing complex memory access patterns. On the other hand, the ASIC simulations of our proposal foretell that it is feasible to achieve an execution time using our proposal in the same order of magnitude than in the local memory scenario. Figure 8 shows the performance loss when remote memory is accessed through the ASIC with a prefetching size of 4 KB compared to the local memory scenario. As can be seen, most of the benchmarks have an execution time that approximately doubles the time required in the local memory case. Nevertheless, it is noteworthy to mention that this slowdown allows to execute these memory-hungry applications in a cluster, which is much cheaper than the initially required mainframe.

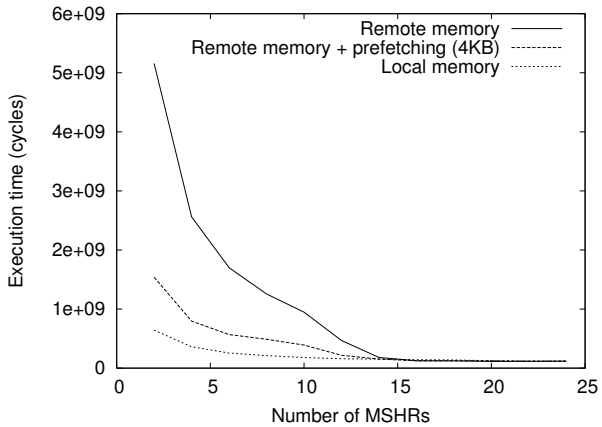
5.2.2 The Importance of the MHA

As mentioned before, the AMD Opteron has 8 MSHRs [17]. The reason is that 8 MSHRs are enough to hide the latency of accessing local memory: before the MHA gets full, the first used MSHR will be released because the memory operation held by it will complete. In this way, the processor will never have to wait due to the MHA. However, when the Opteron is used in a different memory context, the initial size of the MHA may not be enough. In our system, there is some memory accessible by the Opteron that presents much higher latency than expected by the Opteron designers. Therefore, as this is not the scenario initially intended by AMD, the number of required MSHRs may be different.

Figure 9 shows some simulation results. As can be seen, when no prefetching mechanism is used, it is required to increase the amount of MSHRs in order to avoid increasing execution time. Additionally, when no prefetching is used, there is a notorious change in performance between MHA sizes 8 and 10 (STREAM benchmark) and 8 and 14 (random benchmark). This means that we may need up to 14 MSHRs in the worst case scenario in order to effectively mitigate the remote memory latency (if we can get enough memory parallelism out of the code we are executing). However, with the use of prefetching techniques, current Opterons are valid to be used in this new scenario. Nevertheless, it would be worthwhile to have processors with more than 8 MSHRs



(a) STREAM benchmark



(b) Random-STREAM benchmark

Fig. 9 The impact of the MHA

in a remote memory context. However, we cannot expect processor manufacturers to include this change in their designs in the near future due to the fact that a bigger MHA implies higher look up latencies. Thus, mechanisms like prefetching intended to hide remote memory latency are mandatory.

6 Prototyping the New Architecture

We are currently building a 64-node prototype that implements our proposal for non-coherent distributed shared memory. In order to implement the RMC, we are leveraging the HTX card designed by University of Heidelberg [29][21][22]. This card, shown in Figure 10, contains an FPGA where we load several IP blocks also developed by the mentioned university, comprising the Open-Source HyperTransport Core [38], a router for communication among nodes, and the RMC functionality. This prototype will serve as a demonstrator for our proposal.

Fig. 10 HTX card used to implement the RMC

Our prototype is 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. Additionally, this motherboard includes an HTX connector, where we have attached the FPGA previously described.

Regarding the fabric that interconnects all the nodes in the prototype, either an 8x8 2D-mesh or a 4x4x4 3D-mesh can be leveraged. For doing so, we have included a switch at each FPGA that will route the HyperTransport messages exchanged by the RMCs in the cluster. Switching from one topology to the other will be done at boot time just by changing the routing tables and renaming the nodes. Obviously, the fiber cables interconnecting the nodes are not rearranged as this would be extremely costly. Nevertheless, note that a direct network is only one of the feasible interconnects, as the HyperTransport Consortium is currently standardizing other options very interesting, such as HyperTransport over Ethernet and HyperTransport over Infiniband, that will allow the use of standard Ethernet and Infiniband switches.

On the other hand, 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 (in the future we aim to implement the RMC as a regular memory controller). The consequences of this implementation is that Opteron processors in our prototype will 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 eight outstanding requests [26]. Nevertheless, in order to improve the performance of our prototype, we have configured the remote memory ranges

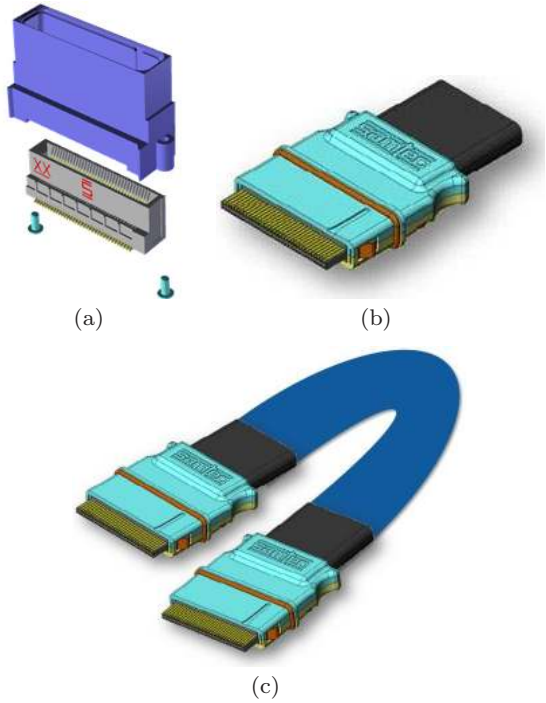


Fig. 11 Female and male cable connectors[3]

as write-back, this is, remote memory blocks can be cached in the processor (just like local memory).

6.1 Making the Idea Low-Cost

In the next sections we will introduce some implementation alternatives aimed at reducing the overall system cost of likely commercial systems based on our proposal. Note that these alternatives are not leverages in our design, as it is a prototype whose design began much before the elements described in this section where standardized.

6.1.1 Motherboard Alternatives

In our prototype cluster the RMC functionality is implemented in an FPGA. This allows fast prototyping and the FPGA reconfigurability enables incremental designs. However, in a commercial implementation it is advisable to migrate to ASIC technology due to economic and performance reasons.

We have three possibilities to connect the RMC to the motherboard. The first one is to develop an ASIC so that it could be placed on the motherboard. The second option is to incorporate the RMC functionality natively into the motherboard chipset, so the cost

[3] Figures 11, 12 and 13 are copyrighted material of the HyperTransport Technology Consortium.

Fig. 12 Vertical HT connectors[3]

of the RMC part would noticeably decrease. Unfortunately, these both solutions require the customization of either the motherboard or the chipset. As such customizations come with high non-recurring engineering costs, it is not likely to happen. Nevertheless, Supermicro has developed a motherboard equipped with a special socket for hosting 3Leaf's Aqua chip. Therefore, a proposal like ours could also be interesting for some motherboard manufacturers.

The third option presented here makes use of an FPGA (or ASIC) equipped add-on card connected to the motherboard via a standard slot connector. The card could be based on the standard HTX slot connector for direct connection to the processor. However, using the HTX connector is not the only choice. For example we could leverage the ubiquitous PCI Express slot connector. Interestingly, the card would use the PCI Express slot purely for power feed and mechanical retention. Instead, for direct CPU connection, the card would leverage some of the new-generation, compact, high-performance HT connectors and cables standardized by the HyperTransport Technology Consortium and productized by Samtec Inc., shown in Figure 11.

These HyperTransport interconnect components are stock components; either in volume production or fully productized, and available from leading interconnect technology vendors. The connector shown in Figure 11(a) enables direct HT link connectivity to processors anywhere on the motherboard, so manufacturers can easily integrate this connector and simplify their motherboard design. Figure 12 shows an example of how flexibly the connector can be positioned on the motherboard, regardless of where the add-on card is physically located. Indeed, the connector can be installed on a CPU Socket

Fig. 13 Right angle HT connectors[3]

module and, in this case, no special motherboard is needed (although one CPU is sacrificed). This new connector decouples the connection to the HT link from the device that is being connected, that is, there is a cable between the motherboard connector and the device. In this way, motherboard manufacturers do not have to allocate space for the device connected to the HT link. It is to be expected that this new versatile connector will be quite appealing to motherboard manufacturers.

Finally, Figure 13 shows the right angle connectors that are used for inter-node connection. This particular connector may be useful for an RMC implementation embedded in the motherboard. In next section we present some alternative inter-node interconnects.

6.1.2 Network Alternatives

According to the HyperTransport Consortium High Node Count specifications [19], HyperTransport messages exchanged between nodes are supposed to use a dedicated network. This means that two independent networks are used in the cluster, the first one for general purposes (common network traffic), and the second one dedicated to the RMCs. In order to comply with the HNC specifications, the add-in cards used in our prototype have six optical fiber connectors that are used to build the dedicated network. This high number of connectors together with the switching and routing functionality included in the FPGA device support direct network topologies like 3D meshes, 6D hypercubes, etc. This makes external and expensive centralized switching components unnecessary. There are commercial systems that leverage this kind of topologies. An example is SGI's Altix Ultraviolet line, based on Intel Nehalem EX processors which either deploys a fat-tree or a 2D torus topology [37].

The first design optimization for our prototype could be the replacement of the optical fiber cables with standard cables, like the ones shown in Figure 11(c). These cables can also be used for inter-node communication. The length limitation of these cables operating at the highest frequency is 2 meters, so the connection scheme of the cluster should take this into account. However, the use of these cables would reduce the interconnection costs.

This system architecture guarantees complete isolation between general traffic and remote memory traffic, and each network can be tailored to the needs of each traffic type. However, for those cases in which the performance of an HT-native network is not mandatory, the HyperTransport Consortium is in the process of standardizing HyperTransport over Ethernet (*HToE*) and HyperTransport over Infiniband (*HToIB*) capabilities, so that the RMC could use this new standards to encapsulate the native HT packets in Ethernet or Infiniband frames [16]. By using one of these techniques, one network is sufficient for both the remote memory and general traffic. However, compared to the performance of a dedicated network, the performance of remote memory accesses will decrease in this case, both due to contention and due to the additional protocol overhead. Also, note that although standard Ethernet or Infiniband switches would be used in these cases, the Ethernet or Infiniband adapters would be different from commodity ones as they should additionally include the RMC functionality (regular adapters should not be used because this would mean generating the HT packets by software, thus noticeably increasing latency and wasting the hardware-only perspective of our approach).

6.1.3 Cluster Configuration

Up to this point we have discussed the interconnect architecture; in this section we will describe the cluster configuration. Our design supports a heterogeneous cluster configuration, with typically different amount of memory and different number of processors for each node. This characteristic allows us to build a cluster with several kinds of nodes, like the example shown in Figure 14. In this case, we use three kinds of processors: we have processing nodes (Magny-Cours processors) that support a higher level of parallelism and that use a larger amount of memory (fast-core processors in Figure 14). This memory may be borrowed from their direct neighbors, that could be nodes containing slower processors and that would mainly act like memory servers (slow-core processors in Figure 14). Moreover, other processors with performance in between the previous two types can be added to the cluster (Shanghai or Barcelona Opterons).

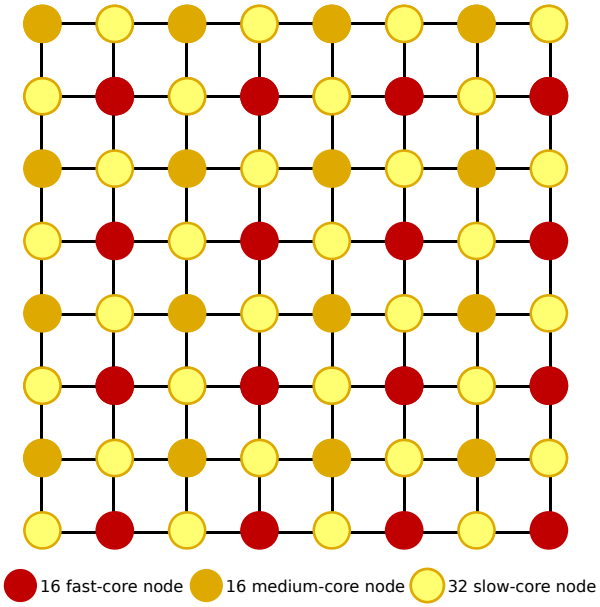


Fig. 14 Example of cluster configuration

This cluster configuration takes into account that two applications with high processing demand will not execute next to each other. This is due to the fact that high processing demand may imply high memory demand, so these applications will expand their memory regions to their neighbors. This way, it is advisable to distribute the high-performance processors across the cluster in some way similar to the one depicted in Figure 14, so that direct access to memory servers can be achieved from computing nodes.

Additionally, this specialization of nodes improves the electric power consumption: there are some nodes whose role in the cluster is memory server, and therefore their processors will only serve as memory controllers. Thus, the processing power can be reduced and therefore the power consumption decreased.

7 Preliminary Prototype Performance

After analyzing the potential of our proposal by simulation, in this section we present some basic performance results from real hardware tests. As described in Section 6, the RMC functionality is currently performed by an FPGA attached to the host through an HTX connector. This kind of implementation allows a quick prototyping of our system but, on the other hand, adds extra latency (and also limits the bandwidth) compared to an ASIC implementation. Thus, the operating frequency of this FPGA also determines the performance of our system.

In order to study the potential of our system and predict the performance trend when improving its implementation, some tests have been carried out and are gathered in Figure 15. The tests mainly consist on changing the speed of the HyperTransport interface. In this way, we can choose between a 400MHz interface (*HT400*) where the HyperTransport link works at 800MT/s, or we can slow down the FPGA so that it uses a 200MHz interface (*HT200*) where the HyperTransport link works at 400MT/s. In both cases, the core logic of the FPGA (RMC) is running at 156MHz. Moreover, to study the scalability of our system, in Figure 15 we have also included results for different distances between the node that executes the analytic benchmark and the node that hosts the memory for that benchmark. Case labeled *0 hops* stands for the loopback mode, that is, the target memory controller for every remote load request is located in the local node, so this message does not cross the external fiber optic link. However, remote load requests use the RMC indeed, but the RMC forwards back the request to a memory controller in the local node. This scenario is useful for studying the overhead related to the link propagation. The analytic benchmark used consists of performing 20 million consecutive accesses to an 8-byte integer array. Data presented in Figure 15 is the averaged time for those accesses.

The first conclusion we can draw from Figure 15 is that the remote load latency increases as the distance between a node and its remote memory increases, as expected. Equation 1 summarizes this behavior.

$$latency = hops * l_{hop} + l_{loopback} \quad (1)$$

where *hops* is the number of nodes between the local node and its remote memory, l_{hop} is the latency added at each hop (it comprises the propagation time through the fiber optic link and also the routing time at the FPGA), and $l_{loopback}$ is a constant time independent of the distance. For example, in the case of *HT400 cachable*, this constant time is 1300ns and l_{hop} is equal to 600ns. As we can see, distance plays an important role in this system, this is, the network topology is a critic characteristic of our proposal. This is especially true when a memory region has been expanded to a big number of nodes, so if we aim to decrease the latency of a remote load we must decrease the average number of hops of the topology, for example, increasing the dimensions of the mesh, moving to a torus topology or even using a indirect topology like a fat-tree.

A second observation according to Figure 15 is that when increasing the HT interface frequency from HT200 to HT400 the latency is reduced only a 20%. This is due to the fact that the HT interface only constitutes a part of the FPGA, and the RMC core functionality keeps its frequency constant as it was previously introduced

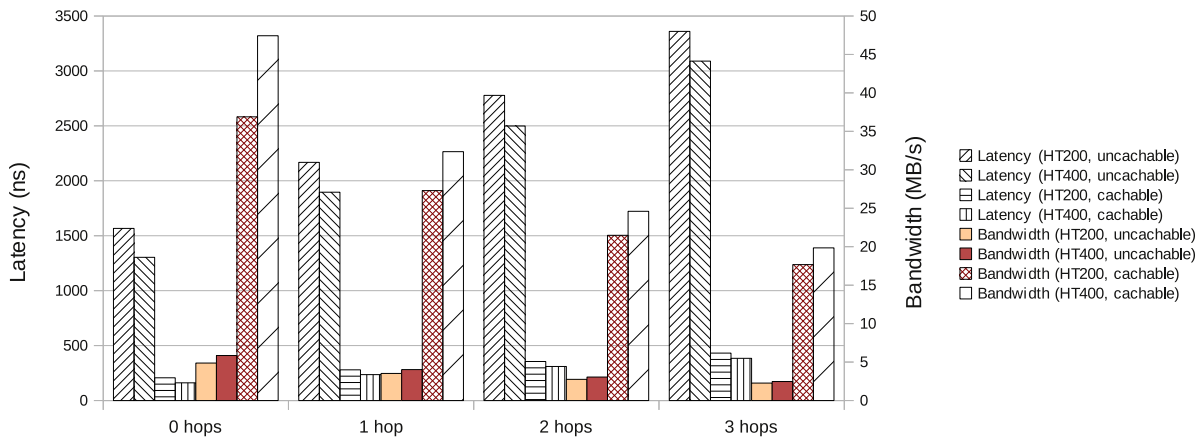


Fig. 15 Analysis on latency and bandwidth scalability

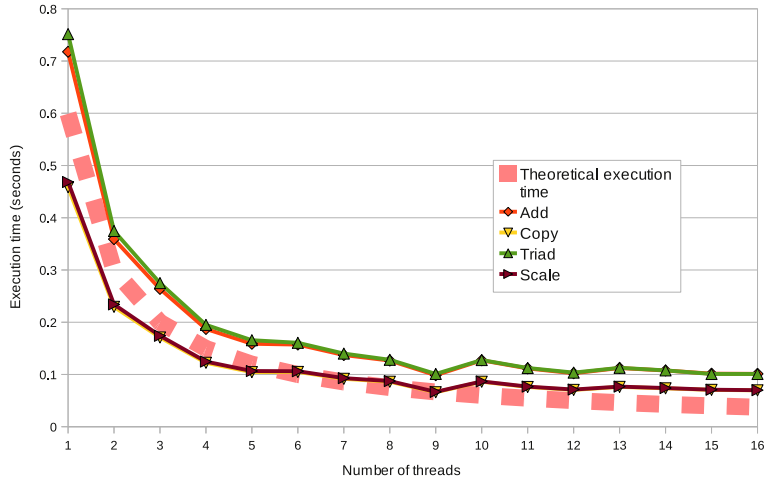
(deeper explanation in next section). One big speedup comes together with the cachable memory attribute: when remote memory is configured as cachable, a remote memory load consists of a 64B packet (one cache line), instead of an 8B packet (size of a CPU register) as in the case of uncachable memory. This means that when accessing consecutive 8B words, only the first one in each cache line will undergo the remote memory latency (naturally, cache reutilization helps to hide remote memory latency).

All the stated behaviors also apply to the bandwidth metric (in inverse proportion). However, there is one fact that may look strange to the reader: although the described system has a very low latency, its bandwidth is not that good. For example, in the 400HT uncachable scenario, the latency is about 1.9ns for one hop distance, but the available bandwidth is 4MB/s, a derisory capacity compared to commercial technologies like Infiband or Ethernet. This low bandwidth is explained by the number of outstanding requests available in the processor. The more outstanding request available the more remote memory operations that can be started in parallel and make the most of the out-of-order execution (as explained in Section 5.2.2). Currently, our design allows only one outstanding request per processor, so each four cores inside a CPU will only be able to launch one remote request at a time. However, this is a system constraint that will be solved in the future. The reason why there is only one outstanding request (instead of eight as an Opteron has at normal work) is because the RMC is currently configured as an I/O unit. The design of the Opteron allows only one outstanding request directed to the I/O space, but we aim to modify the configuration of the RMC so that Opterons could use up to eight outstanding requests.

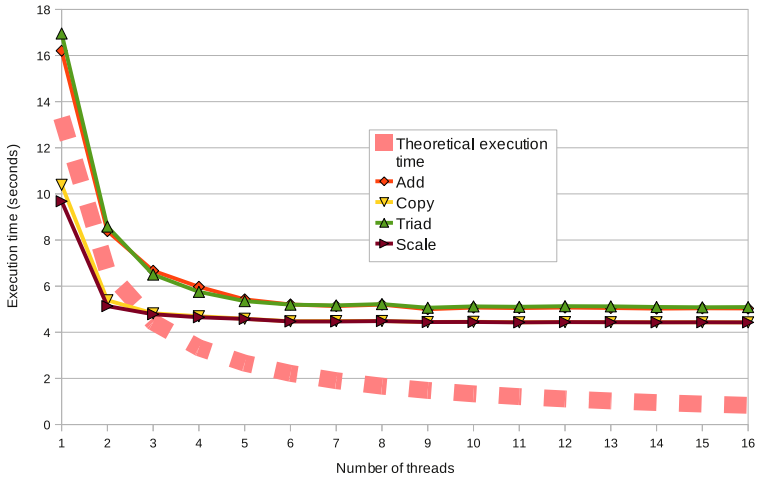
On the other hand, in Figures 16 and 17 we study the scalability of the STREAM benchmark when using our system compared to the local memory scenario. Figure 16 presents execution times for three scenarios: common local memory (not using the RMC) and remote memory both cachable and uncachable for one hop distance. The graphs for each scenario present separately each of the four STREAM operations, as well as the average theoretical execution time. As can be seen, the cachable remote memory presents the worst scalability (the worst time executions obviously correspond to the uncachable remote memory). Regarding the speed-up numbers in Figure 17 we can see how local memory has a maximum speed-up of 7 when using 9 or more threads (the abrupt lines are due to the core/thread affinity OS policy). However, in the remote memory scenario, the maximum speed-up is achieved when using 6 or more cores for the cachable configuration and 5 or more cores for the uncachable one. This time, the speed-ups are 3.2 and 4.5 respectively. In the cachable configuration the different operations in STREAM have more disperse speed-ups because each operation reuses cache lines with a different ratio. Additionally, it is worth explaining that the speed-up for the cachable option is lower because the starting point (execution time with one thread) provided much better results. To sum up, these experiments highlight the importance of having more available outstanding requests, especially when the application uses a high number of cores and intensively accesses memory.

7.1 Performance Trend Prediction

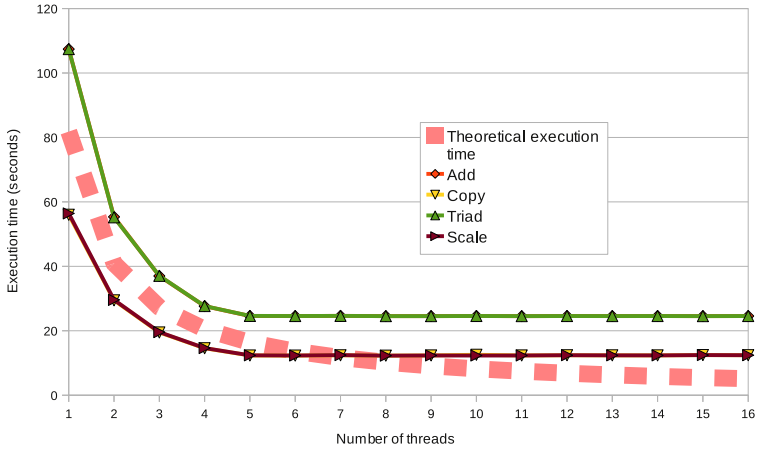
In this section we analyze how the performance of the RMC would benefit from improved implementations.



(a) Local memory

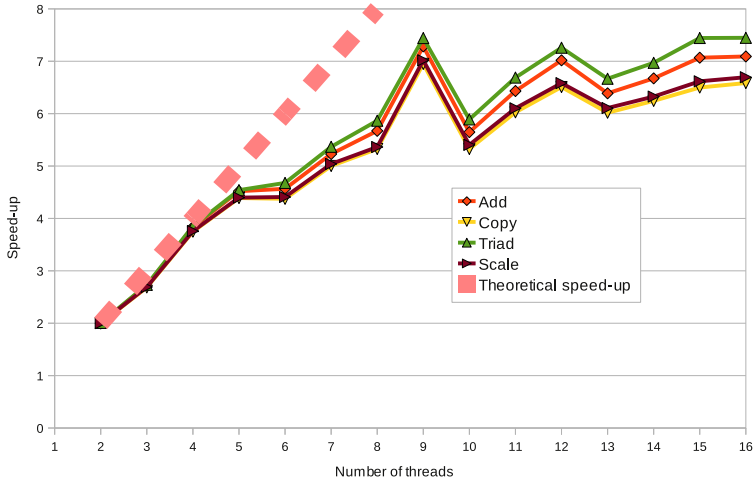


(b) Remote memory, 1 hop (cachable)

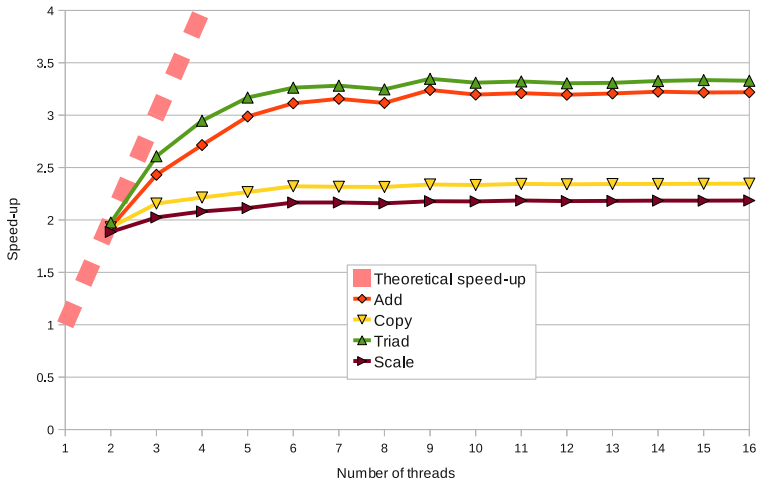


(c) Remote memory, 1 hop (uncachable)

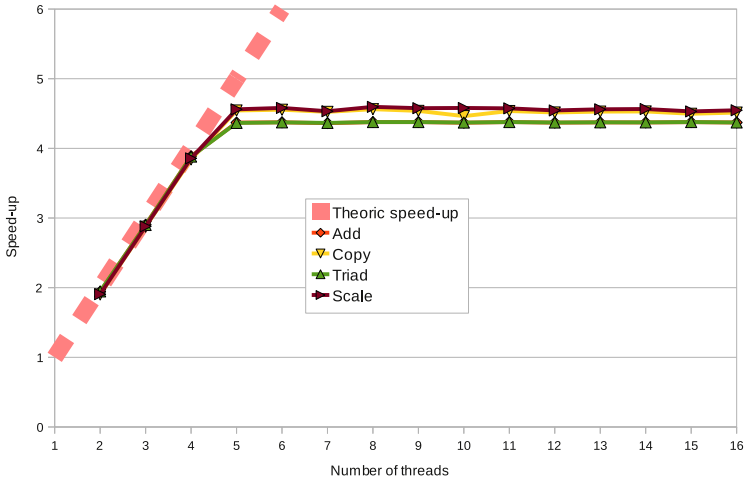
Fig. 16 Execution time for the STREAM benchmark with different number of threads



(a) Local memory



(b) Remote memory, 1 hop (cachable)



(c) Remote memory, 1 hop (uncachable)

Fig. 17 Speed-up for the STREAM benchmark with different number of threads

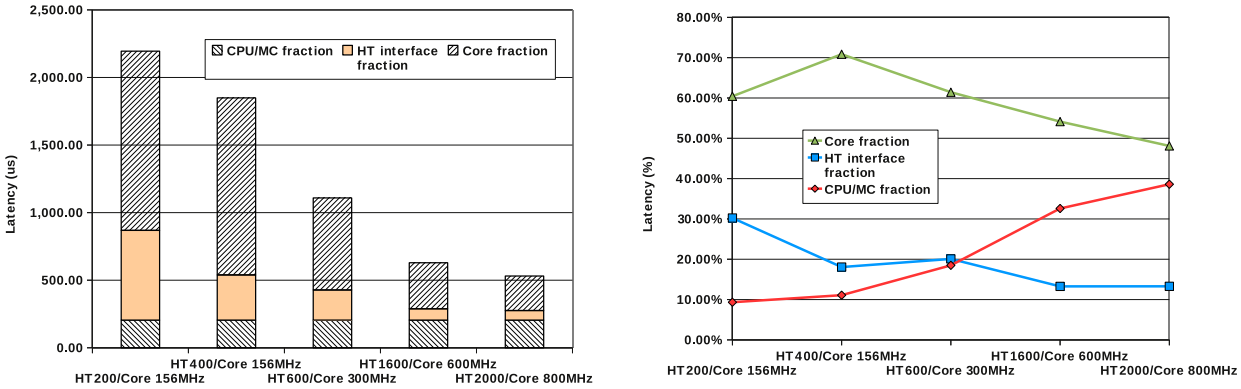


Fig. 18 Latency prediction for remote load operations

The most crucial metric for a performance potential analysis is the remote load latency, as these transactions are typically in the critical loop of execution. The predictions here are based on measurements with HT200/HT400 and a core frequency of 156MHz. The total load latency has been measured, and the fraction of time spent in core logic is determined using results from our simulations with FPGA design software. Counters in the design allow to measure the fraction of time spent on the target side for the memory controller (MC) access. The remaining fraction is the time spent within the source CPU (i.e. software overhead like instruction issue, load/store queue and HyperTransport routing). It is expected that both the MC and the CPU fraction consist of a variable and fixed part: here, the variable part will scale with HT frequency, but the fixed part not (to be more concrete, the fixed part scales with CPU/MC core frequency, which is not varied here). Opposed to this, the complete core logic of the RMC scales linearly with the frequency. With the results from the two experiments, the fixed and variable part of both CPU and MC time can be derived by modeling the variable part in clock cycles, and the fixed part in absolute time. A solution is found if the following equation is valid for both experiments (either for CPU or MC fraction):

$$time = \frac{fixed\_cycles}{HT\_freq + absolute\_time} \quad (2)$$

Now, all fractions of the transaction are modeled: the fixed time spent in the CPU, in the MC, the fraction of time spent on the HT interface and the fraction of time spent in the core logic. Using this, predictions for higher HT frequencies and core frequencies are possible, like it will be the case for possible ASIC implementations: such an implementation is currently expected to reach a core frequency of 800MHz and to implement an HT2000 interface. Then, the full round trip latency of a load transaction decreases down to 534ns.

This reasoning is shown in Figure 18. On the left side of the figure it is shown how remote memory access time noticeably decreases as the implementation of the RMC is gradually improved. On the right side, the relative contribution of each of the components to that latency is depicted. As can be seen, as the implementation of the RMC improves, the contribution of the Opteron itself to the total access time becomes more and more noticeable. Also, note that reducing the contribution of the RMC core to the total latency may be achieved by using wider data paths (this option has not been considered in Figure 18, which has been carried out by considering a 32-bit wide data path, but wider paths may be feasible, like using 64 bits).

8 Future Work

Results presented in Section 5 have shown the feasibility of our proposal for non-coherent shared-memory clusters. These results have been later supported by Section 7. However, they are the first step in fully evaluating this new approach to shared memory. Further work is required to better assess the benefits of our proposal. To do so, we are planning to analyze it by using real applications in the simulations. Also, as our proposal creates a very non-uniform memory architecture, more work is required to develop and/or adapt techniques like prefetching or other mechanisms, in order to alleviate the high latencies when accessing remote memory. Additionally, further research is required on how to combine improved prefetching mechanisms with a private RMC cache.

As mentioned before, we are currently building a 64-node prototype that will implement all the features

described in this paper. For this prototype to be finished, there are many secondary concerns to deal with: memory allocation policies, operating system capabilities, virtualization issues, security, etc. Initial performance numbers from real executions in this prototype show that it properly works and that remote memory access latency is affordable.

Moreover, the prototype can be tested with other applications. We are willing to do so with two main application domains: virtualization layers and databases. On one hand, we aim to integrate our proposal in a virtualized environment and analyze the service that a virtualization layer can offer to the upper operating systems. On the other hand, databases constitute an extraordinary opportunity where new usages of remote memory become interesting. A part or the entire database can be loaded into the cluster memory so that each node allocates a piece of the data in its local memory. Unlike MySQL Cluster [36], each node can access local or remote data indistinctively. Therefore, a query that requires concurrent access to data spread across several nodes can be executed at a very high speed. As no coherency is provided by our proposal among the nodes of the cluster, read-only mode must be set for the shared memory. Nevertheless, this is still very useful for static or non-frequently updated databases like the ones present in many servers in the Internet.

Finally, a natural step forward is to provide interprocess inter-node communication mechanisms so that processes in different nodes (and different OSes) could synchronize or exchange data. As no coherency is maintained among nodes, explicit software coherency operations should be used. This communication paradigm allows the coherency protocol to be tailored to the application needs and not to maintain coherency where it is not required.

9 Conclusions

In this paper we have presented a low-cost approach to efficiently implement shared memory across a cluster. Our proposal splits the cluster into non-overlapping coherence domains that may span to several computers. Nevertheless, as coherency domains are limited to the processors in a single motherboard, the memory granted to a given process in the cluster can be increased without expecting a performance overhead due to the coherency protocol. The straight-forward use of such system is to speed up the execution of applications with a memory footprint larger than the memory available in a single computer, although our proposal can be used in other contexts.

Both simulation and real results have shown the feasibility of the proposed system, which completely relies on hardware to access remote memory. This is accomplished by leveraging the HyperTransport protocol. Further research is required to minimize the impact on performance of the larger access time to remote memory. These optimizations will deliver interesting results in the 64-node prototype we are currently building to demonstrate this technology.

Acknowledgements This work has been supported by PROMETEO from Generalitat Valenciana (GVA) under Grant PROMETEO/2008/060.

References

1. 3leaf Systems. http://www.3leafsystems.com
2. Dynamic Logical Partitioning. White Paper. http://www.ibm.com/systems/p/hardware/whitepapers/dlpar.html
3. Gaussian 03. http://www.gaussian.com
4. IBM z Series. http://www.ibm.com/systems/z
5. In-Memory Database Systems (IMDSs) Beyond the Terabyte Size Boudary. http://www.mcobject.com/130/EmbeddedDatabaseWhitePapers.htm
6. MBA3 NC Series Catalog. http://www.fujitsu.com/global/services/computing/storage/hdd/ehdd/mba3073nc-mba3300nc.html
7. NUMAChip. http://www.numachip.com/
8. ScaleMP. http://www.scalemp.com
9. Violin Memory. http://violin-memory.com
10. HyperTransport Technology Consortium. HyperTransport I/O Link Specification Revision 3.10 (2008). Available at http://www.hypertransport.org
11. Acharya, A., Setia, S.: Availability and Utility of Idle Memory in Workstation Clusters. SIGMETRICS Perform. Eval. Rev. **27**(1), 35–46 (1999). DOI http://doi.acm.org/10.1145/301464.301478
12. Anderson, T., Culler, D., Patterson, D.: A case for NOW (Networks of Workstations). Micro, IEEE **15**(1), 54–64 (1995). DOI 10.1109/40.342018
13. Bienia, C., Kumar, S., et al.: The parsec benchmark suite: Characterization and architectural implications. In: Proceedings of the 17th PACT (2008)
14. Chapman, M., Heiser, G.: vNUMA: A virtual shared-memory multiprocessor. In: Proceedings of the 2009 USENIX Annual Technical Conference, pp. 349–362. San Diego, CA, USA (2009)
15. Charles, P., Grothoff, C., Saraswat, V., et al.: X10: an Object-Oriented Approach to Non-Uniform Cluster Computing. SIGPLAN Not. **40**(10), 519–538 (2005). DOI http://doi.acm.org/10.1145/1103845.1094852
16. Consortium, H.: HyperTransport High Node Count, Slides. http://www.hypertransport.org/default.cfm?page=HighNodeCountSpecification
17. Conway, P., Hughes, B.: The AMD Opteron Northbridge Architecture. IEEE Micro **27**(2), 10–21 (2007). DOI http://dx.doi.org/10.1109/MM.2007.43
18. Conway, P., Kalyanasundharam, N., Donley, G., et al.: Blade Computing with the AMD Opteron Processor (Magny-Cours). Hot chips 21 (2009)

19. Duato, J., Silla, F., Yalamanchili, S., et al.: Extending HyperTransport Protocol for Improved Scalability. First International Workshop on HyperTransport Research and Applications (2009)
20. Feeley, M.J., Morgan, W.E., Pighin, E.P., Karlin, A.R., Levy, H.M., Thekkath, C.A.: Implementing global memory management in a workstation cluster. In: SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles, pp. 201–212. ACM, NY, USA (1995). DOI http://doi.acm.org/10.1145/224056.224072
21. Fröening, H., Litz, H.: Efficient Hardware Support for the Partitioned Global Address Space. In: 10th Workshop on Communication Architecture for Clusters (2010)
22. Fröening, H., Nuessle, M., Slogsnat, D., Litz, H., Brüening, U.: The HTX-Board: A Rapid Prototyping Station. In: 3rd annual FPGAworld Conference (2006)
23. Garcia-Molina, H., Salem, K.: Main Memory Database Systems: an Overview. Knowledge and Data Engineering, IEEE Transactions on **4**(6), 509–516 (1992). DOI 10.1109/69.180602
24. Gray, J., Liu, D.T., Nieto-Santisteban, M., et al.: Scientific Data Management in the Coming Decade. SIGMOD Rec. **34**(4), 34–41 (2005). DOI http://doi.acm.org/10.1145/1107499.1107503
25. IBM journal of Research and Development staff: Overview of the IBM Blue Gene/P project. IBM J. Res. Dev. **52**(1/2), 199–220 (2008)
26. Keltcher, C., McGrath, K., Ahmed, A., Conway, P.: The AMD Opteron Processor for Multiprocessor Servers. Micro, IEEE **23**(2), 66–76 (2003). DOI 10.1109/MM.2003.1196116
27. Kottapalli, S., Baxter, J.: Nehalem-EX CPU Architecture. Hot chips 21 (2009)
28. Liang, S., Noronha, R., Panda, D.: Swapping to Remote Memory over InfiniBand: An Approach using a High Performance Network Block Device. In: Cluster Computing, 2005. IEEE International, pp. 1–10 (2005). DOI 10.1109/CLUSTR.2005.347050
29. Litz, H., Fröening, H., Nuessle, M., Brüening, U.: A HyperTransport Network Interface Controller for Ultra-low Latency Message Transfers. HyperTransport Consortium White Paper (2007)
30. Litz, H., Fröening, H., Nuessle, M., Brüening, U.: VELO: A Novel Communication Engine for Ultra-Low Latency Message Transfers. In: Parallel Processing, 2008. ICPP '08. 37th International Conference on, pp. 238–245 (2008). DOI 10.1109/ICPP.2008.85
31. Magnusson, P., Christensson, M., Eskilson, J., et al.: Simics: A Full System Simulation Platform. Computer **35**(2), 50–58 (2002). DOI 10.1109/2.982916
32. Martin, M., Sorin, D., Beckmann, B., et al.: Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. SIGARCH Comput. Archit. News **33**(4), 92–99 (2005). DOI http://doi.acm.org/10.1145/1105734.1105747
33. McCalpin, J.D.: Memory Bandwidth and Machine Balance in Current High Performance Computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter pp. 19–25 (1995)
34. Oguchi, M., Kitsuregawa, M.: Using Available Remote Memory Dynamically for Parallel Data Mining Application on ATM-connected PC Cluster. In: IPDPS 2000. Proceedings. 14th International, pp. 411–420 (2000). DOI 10.1109/IPDPS.2000.846014
35. Oleszkiewicz, J., Xiao, L., Liu, Y.: Parallel Network RAM: Effectively Utilizing Global Cluster Memory for Large Data-Intensive Parallel Programs. In: Parallel Processing, 2004. ICPP 2004. International Conference on, pp. 353–360 vol.1 (2004). DOI 10.1109/ICPP.2004.1327942
36. Ronstrom, M., Thalmann, L.: MySQL Cluster Architecture Overview. Technical White Paper. MySQL (2004)
37. SGI: Technical Advances in the SGI Altix UV Architecture, White Paper. `http://www.sgi.com/products/servers/altix/uv/`
38. Slogsnat, D., Giese, A., Nüssle, M., Brüning, U.: An Open-source HyperTransport Core. ACM Trans. Reconfigurable Technol. Syst. **1**(3), 1–21 (2008). DOI http://doi.acm.org/10.1145/1391732.1391734
39. Szalay, A.S., Gray, J., vandenBerg, J.: Petabyte Scale Data Mining: Dream or Reality? CoRR **cs.DB/0208013** (2002)
40. Tuck, J., Ceze, L., Torrellas, J.: Scalable Cache Miss Handling for High Memory-Level Parallelism. Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on (2006)
41. Yelick, K.: Computer architecture: Opportunities and challenges for scalable applications. Sandia CSRI Workshop on Next-generation scalable applications: When MPI-only is not enough (2008)
42. Yelick, K.: Programming models: Opportunities and challenges for scalable applications. Sandia CSRI Workshop on Next-generation scalable applications: When MPI-only is not enough (2008)