Getting Rid of Coherency Overhead for Memory-Hungry Applications

Héctor Montaner*, Federico Silla*, Holger Fröning†, and Jose Duato*
*Universitat Politècnica de València, Departament d'Informàtica de Sistemes i Computadors
e-mail: hmontaner@gap.upv.es, {fsilla,jduato}@disca.upv.es
†University of Heidelberg, Computer Architecture Group
e-mail: froening@uni-hd.de

***Abstract*—Current commercial solutions intended to provide additional resources to an application being executed in a cluster usually aggregate processors and memory from different nodes. In this paper we present a 16-node prototype for a shared-memory cluster architecture that follows a different approach by decoupling the amount of memory available to an application from the processing resources assigned to it. In this way, we provide a new degree of freedom so that the memory granted to a process can be expanded with the memory from other nodes in the cluster without increasing the number of processors used by the program. This feature is especially suitable for memory-hungry applications that demand large amounts of memory but present a parallelization level that prevents them from using more cores than available in a single node.**

The main advantage of this approach is that an application can use more memory from other nodes without involving the processors, and caches, from those nodes. As a result, using more memory no longer implies increasing the coherence protocol overhead because the number of caches involved in the coherent domain has become independent from the amount of available memory.

The prototype we present in this paper leverages this idea by sharing 128GB of memory among the cluster. Real executions show the feasibility of our prototype and its scalability.

I. Introduction

Current proposals for high performance computing and data centers are usually based on the deployment of commodity x86-based servers assembled into racks. The reason for building large systems in this way is simple: cost. Effectively, although x86-based clusters may be neither the best architectural choice for building large systems nor they may provide the best performance, their cost has made them the default choice for achieving large computing power, although that computing power is not provided by a single system but by the aggregation of smaller and independent computers.

The reason for the good cost/performance ratio of clusters is that they are made of components that are manufactured in large volumes, thus recovering their design and production costs. Additionally, as these commodity components are noticeably more cost-effective for manufacturers, they try to focus their road maps on them, likewise reducing their cost, thus closing the loop.

Clusters are based on relatively inexpensive coherent shared-memory machines that are able to scale up to 64 computing cores by using the last developments by AMD [1] or Intel [2]. Additionally, their memory capacity may be as high as a few hundred Gigabytes[1]. However, 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. This characteristic has several consequences. On one hand, cluster administrators provision each of the computers in the cluster for its worst-case memory usage, what usually leads to memory sizes much larger than required for most applications [3]. On the other hand, and apparently contradictory to the previous statement, there may be an unbalance between the amount of computing resources and memory resources present in a motherboard. This unbalanced configuration is due to the fact that most shared-memory applications are not able to exhaust the total amount of cores present in a motherboard while they easily outgrow the available memory in a single node [4][5][7].

In order to provide the flexibility required to deal with applications demanding very different amounts of memory, several solutions have been devised, like the Aqua chip by 3Leaf [6], that provides a coherent distributed shared-memory system across the cluster glueing together computing cores and memory resources. However, it does so by paying 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 Aqua chip may allow applications to span to as many cores as required and to use as much memory as needed. Nevertheless, as many applications do not scale beyond the number of cores found in a single motherboard (although they may still benefit from the large amount of memory present in a coherent distributed shared-memory cluster), 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 reason is that all the caches involved in the execution of that application remain in the same motherboard, and thus there is no need to propagate coherency operations (e.g. probes and invalidations) to caches out of the node where

[1]This limitation is imposed by motherboard manufacturers.

the coherency operation started. It can be seen that, in these cases, an aggregation technique like the Aqua chip may be counterproductive because of the overhead of the inter-node coherency protocol. There is, therefore, a need for decoupling processor aggregation from memory aggregation.

In this paper we present a prototype that features a non-coherent distributed shared-memory architecture in clusters, thus avoiding the penalty due to the inter-node coherency protocol. Our proposal dynamically partitions the cluster into non-overlapping coherent domains, each of them containing the cores and caches of a single motherboard and perhaps spanning to memory located in other motherboards.

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

II. Related Work

Disk swapping is the traditional approach for getting additional memory when the available physical memory exhausts. However, when the working set of an application is bigger than the available memory, the trashing problem easily arises, increasing execution time to prohibitive levels.

Remote swap [7][8] is a similar technique that 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. Actually, 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 [9]. However, remote swap still suffers from the operating system (OS) overhead.

A different approach is followed by Violin Memories, that offers a memory server that can hold up to 504 GB of RAM [10]. Unfortunately, this is an expensive approach that additionally presents a large memory access time (3 microseconds) because the OS is involved in every memory access. Moreover, it is an expensive approach (a server populated with only 120 GB costs more than $20.000). On the other hand, [11] proposes a very similar approach, but from a theoretical perspective.

Other proposals have been made in order to reduce the unbalance between computing and memory resources. For example, the use of memory compression has been proposed in [12][13]. Moreover, replacing conventional DRAM by more compact, but slower, storage devices, like NAND flash, has been proposed by companies like Virident [14] and Texas Memory [15].

Some companies, like 3Leaf [6], ScaleMP [16], and Numascale [17] provide more resources to applications by aggregating the processors and/or memory in a cluster into a single computer. In the case of ScaleMP, a virtualizing software layer aggregates multiple x86 systems into a single virtual symmetric multiprocessor where coherency is maintained among the integrating computers. However, its main drawback is that it is software based, thus reducing performance. 3Leaf and Numascale follow a similar approach but from a hardware-based perspective. Nevertheless, as coherency must be maintained throughout a large number of computers, the scalability and performance of these proposals are limited in practice.

A different proposal from industry is IBM's Dynamic Logical Partition (DLPAR) [18], 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.

III. 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 single 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.

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:

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

A. System Overview

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

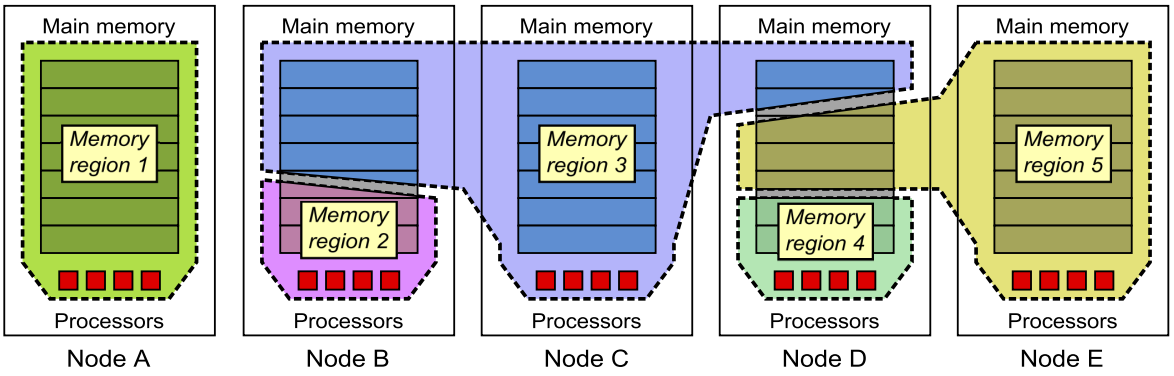


Fig. 1. An example of memory sharing among the nodes of a cluster

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.

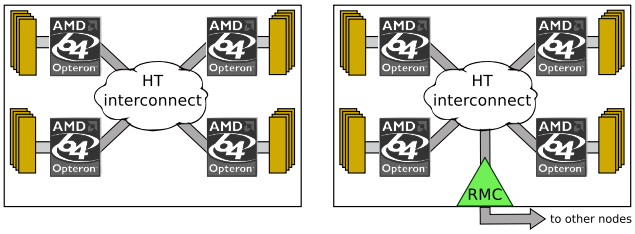
It is important to emphasize that as memory regions are independent, processes in node A can only access region 1, processes in node B can only access region 2, processes in node C can only access region 3, etc. In the same way, as all processors in a node can only access one (the same) memory region, 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 therefore applications do not undergo 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.

B. System Architecture

HyperTransport technology [19] 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 [20], 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 implement our proposal, which involves creating a new hardware component that we will refer to as *Remote Memory Controller* (RMC). This new component will be presented to the processors in the motherboard as a HyperTransport I/O Unit as shown in Figure 2(b), which means that the RMC is out of the coherent system of the motherboard (in the future, we aim to implement the RMC as a regular memory controller inside the coherent system). The RMC will forward local HT transactions to remote nodes, where they are handled by the memory controllers of the remote CPUs. So, the RMC has no memory banks directly attached to it, but relies on the memory banks installed in other nodes in the cluster. If a local process



(a) (b)

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

wants to access remote memory, it just needs to perform a load or store operation on the memory region mapped to the RMC.

Figure 3 is a representation of the shared memory distribution seen by every node in a 16-node example cluster. In this example, each node has 4 sockets, each of them attached to 4GB of main memory. Nevertheless, the node can see its own memory, 16GB, plus the memory in the nodes of the cluster (including its own memory again), 256GB. This is because above 16GB, the memory is mapped to the RMC, as shown in the right column of Figure 3. Note that although the entire memory can be addressed, some memory in each node is used by the local OS and, therefore, this memory should never be used by the rest of the nodes. As will be explained in Section IV, in our particular implementation, 8GB out of the 16GB present in each node are reserved for private use and the other 8GB are aggregated to form a 128GB shared memory pool across the cluster.

As can been seen in Figure 3, 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 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.

Note that node identifiers start at node number 1, this is, 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

[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.

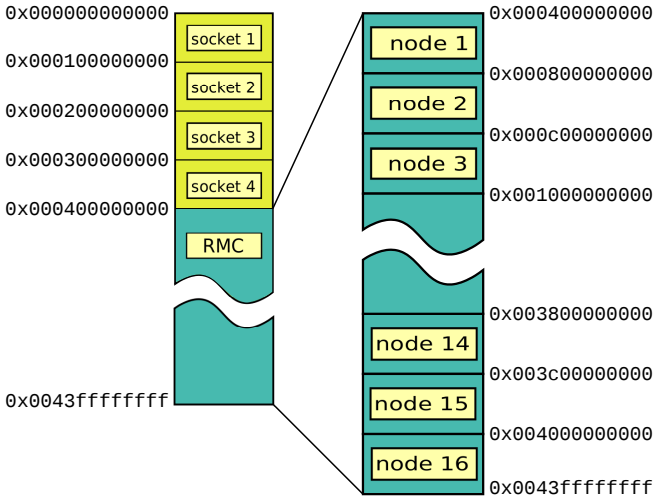


Fig. 3. Example of the memory map of a node in a 16-node cluster.

translation table. Unfortunately, there will be an overlapped segment in the memory map for each node. For example, if node 1 addresses memory between `0x000400000000` and `0x0007ffffffff`, it will be referring to its own local memory (loopback mode). However, this will never happen in practice because of 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.

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

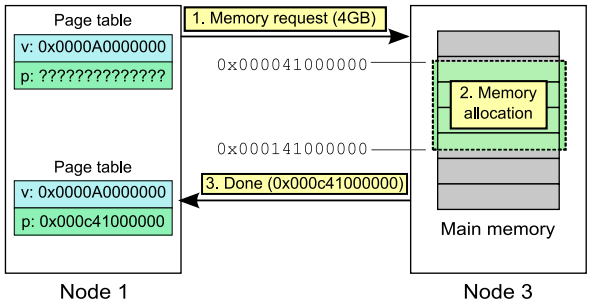


Fig. 4. Node 1 reserves remote memory in node 3

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. However, the presented reservation mechanism requires that the memory reserved in a remote node is not swapped to disk, what could convert out idea in some kind of remote swapping mechanism. Nevertheless, current OSes already allow marking some memory pages as not swappable to disk. On the other hand, 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.

IV. Prototyping the New Architecture

In this section we present the 16-node prototype we have built to demonstrate the feasibility of the proposed architecture. We first present the way we have implemented the RMC. Then we introduce our prototype.

A. Implementing the RMC

AMD uses HyperTransport to interconnect the different memory controllers in a motherboard. As the RMC is presented to the rest of the processors as a HyperTransport device that can access some amount of memory, its design requires providing it with a HyperTransport interface so that it can communicate with the memory controllers 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. Therefore, for communication among nodes, we will make use of the High Node Count HyperTransport specification 1.0 recently published [21], which extends

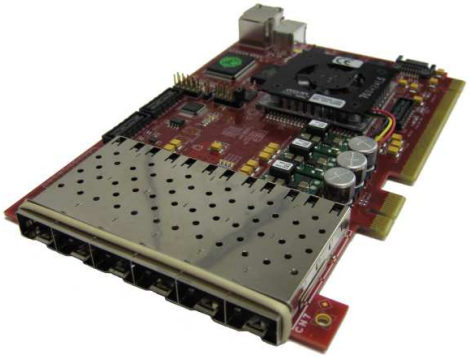


Fig. 5. HTX card used to implement the RMC

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 [21] 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 [21] 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. 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. Actually, our prototype uses the HTX card designed by University of Heidelberg [22][23][24]. This card, shown in Figure 5, contains an FPGA where we load the Open-Source HyperTransport Core [25] and the RMC functionality.

B. Building the Cluster Prototype

We have built a prototype that implements our proposal for non-coherent distributed shared-memory. Our prototype consists of 16 nodes based on the Supermicro H8QM8-2+ motherboard containing four 2.1GHz quad-core Opteron processors. Each processor is attached 4GB of 800MHz DDR2 memory. Thus, each node features 16 cores and 16GB of main memory. Additionally, this motherboard includes an HTX connector, where we have attached the FPGA previously described. Each operating system is booted with only 8GB so that the other 8GB are set aside for the shared memory pool, that adds up to 128GB.

Regarding the fabric that interconnects the 16 nodes of the prototype, a 4x4 2D-mesh will 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. Additionally, we have made use of four of the six connectors included in the HTX card mentioned above in order to build the mesh topology. Nevertheless, note that a direct network is only one of the feasible interconnects, as the HyperTransport Consortium is currently standardizing other option 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 [20]. Nevertheless, in order to improve the performance of our prototype, we have configured the remote memory ranges as write-back, this is, remote memory blocks can be cached in the processor (just like local memory). However, as coherency is not maintained in I/O memory, we are restricted to use only serial applications and bind the process to a single core. Note that when there is a read-only phase in the application, we can successfully parallelize it and execute it with several threads, as no coherency is needed (once the cache contents corresponding to the write phase have been flushed).

Finally, let us explain how the remote memory is actually granted to the application so that the reader can have a practical view of the system. We take an application coded in C or C++ and already compiled (the only restriction is that the linkage must not be static). This way, with no need for recompiling it we interpose our library that implements a special *malloc* and *free* functions. Thus, every time the application tries to allocate dynamic memory, we will intercept it, then we allocate remote memory, and finally we return to the application a pointer to a virtual address mapped to remote memory. As can be seen, this process is completely transparent to the application. Note that we use this library only to catch memory-reservation operations, and it is important to emphasize that the subsequent load and store operations over that memory will be regular memory instructions where no software layers are involved, as explained before.

V. Evaluating the Prototype

In this section we present some performance metrics that show the current state and capabilities of our 16-node prototype. To do so, we first analyze which is the remote access latency in the new architecture. Then we compare its performance with that of a single machine populated with 128GB of local memory, thus avoiding the penalty of remote accesses. We will also compare the behavior of our prototype with the remote swap technique, that has been widely discussed in research bibliography [26][27][8]. The configuration of the

remote swap scenario comprises two nodes connected through Gigabit Ethernet. The client node has 256MB of main memory, as it will be easier to study the swap overhead. The operative system (Linux kernel version 2.27) has a memory footprint of 100MB, so there are roughly 156MB of free local memory. We use the Network Block Device (NBD) [28] for automating the remote swap process. NBD is is a Linux implementation of Network Block Device over TCP/IP using kernel-level socket interface for network communication.

A. Performance Analysis

The RMC is designed to transmit load and write operations from one computer to another. Nevertheless, the latency between nodes is not critical for the write operation as it can be a posted write, that is, no response is required so there is no need for the processor to wait until the operation reaches the remote node. For this reason, we will focus on the load operation as this will be the key metric in subsequent studies.

As the proposed architecture is intended to build a cluster with a large number of nodes, in the first test we have analyzed the time required to access a memory server located several hops away in the 2D-mesh. To do so, we have used a benchmark that access 10 million random positions of an array mapped to the memory of the remote node. Figure 6 shows the execution time of such a benchmark when the distance to the memory server increases. Note that in our 2D-mesh the maximum number of hops is 6.

As can be deduced from the figure, each remote access takes 2.2 μs to complete when the memory server is one hop away. This time comprises the time required for generating the random location where to access, decoding and issuing the load instruction, sending a read request to the local RMC through the on-board HyperTransport connection, routing the packet at the switch in the local RMC, crossing the optical fiber to the other node, processing the message by the remote RMC, accessing the memory controller in the remote node, getting the data from RAM and all the way back to the original processor cache. When the generation of random numbers is avoided by using a fixed stride, the remote latency is reduced to 1.9 μs. Moreover, each additional hop adds a 600 *ns* latency to the load operation (300 *ns* in each direction). Thus, the latency of the memory access linearly increases with the distance between the nodes.

Regarding the bandwidth, as we have mentioned before, we are limited to one outstanding load operation per core, so the inter-node bandwidth is constrained at the requesting processor. However, we can study this feature by running several threads, one in each core of the motherboard. Figure 7 presents the results for experiments with different number of threads.

The first group of results on the left (1 server) shows how the execution time of the random benchmark varies when increasing the number of threads. As expected, when the benchmark uses two threads, the required time for executing a fixed amount of accesses becomes half the time, because each thread performs in parallel half of the total accesses. This intuitive rule does not apply to the case of four threads, as we

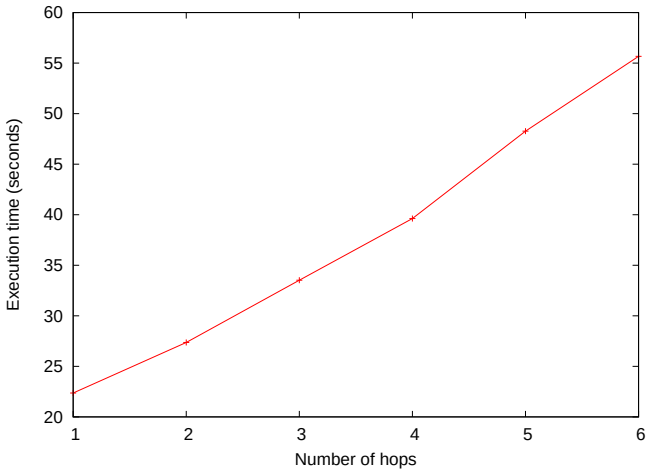


Fig. 6. Remote memory latency proportionally increases with the distance between client and server

can see how the time does not get reduced in the expected proportion. From these results we can deduce that there is a bottleneck either in the local RMC or in the remote RMC. This bottleneck causes that the system saturates with the load request rate generated by two threads.

In order to locate the bottleneck, we have re-run the benchmark but with a different system configuration: this time the remote memory will be distributed over four memory servers instead of one (right side of Figure 7). If the memory server was congested in the previous configuration, now, as it is replicated four times, we should see how four threads work better than two. However, as no speed up is achieved (bar labeled ”4t, 1 hop” on the right side of the figure), we know that the remote server is not the bottleneck. Therefore, the bottleneck is located in the local RMC. To check this, we have re-run the benchmark but moving away the memory servers from the client node. In Figure 7 we can see that when the servers are two and three hops away from the client, the execution time slightly decreases. Although the time difference is quite small, it was expected the opposite behavior, that is, a noticeable increase in the execution time, as it was previously showed, because of the increased distance. This unforeseen behavior means that the bottleneck is in the local RMC: as the read operations have to travel a longer path (i.e. higher latency), the memory request rate generated by four threads is lower than before (only one outstanding request), and the local RMC has more time to process the load operations and does not become a bottleneck.

The next set of tests is intended to further analyze a possible bottleneck in the server side of the architecture. This time, there is only one memory server and we will increase the number of clients (nodes and threads) that access it. Figure 8 shows the results for these experiments. The execution time shown in this figure corresponds to a control thread running in a node that is directly connected to the server by a link only used by them, so that traffic congestion in the network will not affect this thread. In this way, we will analyze how congestion in the server due to the accesses of the other clients in other

Fig. 7. Analyzing the client performance

nodes affects the control thread.

We can see in Figure 8 that even when three additional nodes with four threads per node are stressing the server, the time required by the control thread to complete the benchmark does not vary. But when the number of clients and threads continues increasing we can see how the execution time gets worse, not as a result of network congestion but as a result of RMC congestion in the server. There is another interesting result: note that in the previous experiment, with more than two threads per client the RMC in client role got saturated. However, now the number of memory requests that arrive to the server increases when increasing the number of threads in the clients, even beyond two threads. The explanation for this is the same as in the previous experiment: when the network is congested, the latency of the load operations increases and this alleviates the bottleneck at the client RMC as now it has more time to process the requests.

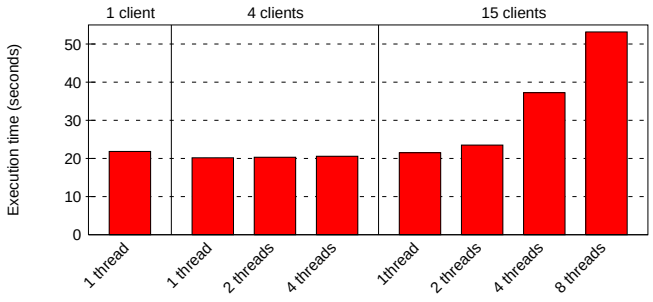


Fig. 8. Analyzing the server performance

B. Comparative Analysis of Data Retrieval Operations

In this section, we analyze the performance of our prototype by stressing remote memory through a data retrieval operation that mimics database searches. We also compare the results with the performance achieved by the remote swap technique.

This test is based on an ordered list of keys. The operation of finding an element (i.e. a key) in this list can be achieved in $O(\log_2 n)$ comparisons by using, for example, binary search. However, when the list is stored in a secondary memory, for example a disk. The reason is that this method requires an unnecessary large amount of time. When data is retrieved from disk or from remote memory through remote swap, the first access to a memory page takes a noticeable time while subsequent accesses to that page have a lower latency as the page has been moved to local main memory. As can be seen, locality is crucial when dealing with this kind of secondary storages. Unfortunately, the way a binary search proceeds will ask for memory positions widely separated: first, it will require the key located at the middle of the array, then it will jump to the middle of one of the sub-arrays, etc. So, if the list comprises a big number of memory pages, most of them will be only retrieved for just one access (except the last pages). This is the reason why databases and file-systems do not use a binary search tree but a generalization: *b-tree*[3].

A b-tree is a tree data structure where each node contains a sorted list with an arbitrary number of keys, K. These keys delimit the range of the keys that each sub-tree contains, this is, sub-tree s_n can only contain keys between k_{n-1} and k_n. There are also two sub-trees that contain keys lower than the first key in the list and keys higher than the last key in the list, respectively; this way, each internal node has up to $K+1$ children.

The method for finding an element in a b-tree is similar to the case of a binary tree, in the sense that $\log_m N$ nodes must be visited, where N is the total number of nodes in the b-tree and m is the number of children that an internal node has. The difference is that in each node the array of keys has to be searched in order to find the appropriate child where to continue the retrieval operation (the binary tree has only one key per node so there is no need for searching). However, as the search operation inside each node can be done by binary search, the total cost of retrieving one element in the b-tree is still $O(\log_2 n)$ comparisons, as shown in [29].

However, this data structure allows a higher access locality than a binary tree, because the sorted array inside the nodes can be placed in a single memory page (as long as the size of the array does not exceed the page size). Therefore, the more keys a node has the less nodes the algorithm has to visit, because the tree has lower depth. If we assume that visiting a new node means retrieving a new memory page, then we are interested in reducing the tree height as much as possible in the remote swap scenario. Databases implementations try to maximize the number of comparisons within a page before being forced to retrieve a new page. This is accomplished by allocating as much keys per node as keys can be fitted in a page. This way, the b-tree search operation performance is maximized. Equation 1 describes the memory time for an application that uses remote swapping.

$$T_{remote\_swap} = A_{total} * L_{local} + \frac{A_{total}}{A_{page}} * L_{swap} \quad (1)$$

where A_{total} is the total amount of memory accesses throughout the execution, A_{page} is the number of accesses to a page

[3]Note that in-memory databases usually implement hash indexes, as this structure presents even better performance when it is stored in memory. Thus, by using b-trees in this study, we relinquish the advantage over remote swap provided by hash indexes when used in remote memory.

(during its lifetime in main memory), L_{local} is the latency of accessing RAM memory in the motherboard, and L_{swap} is the latency of retrieving a page from remote memory. Equation 2 describes the memory time for an application that uses our remote memory architecture.

$$T_{remote\_memory} = A_{total} * L_{remote} \quad (2)$$

where L_{remote} is the latency of retrieving one memory line from remote memory. As can be seen, the remote memory system is less sensitive to locality than remote swap, as the number of pages hit by the application has no influence in its execution time (keeping constant the number of accesses).

To carry out a fair comparison, we should first discover the number of children that minimizes the search time when the b-tree is stored in remote swap. As that number depends on the b-tree implementation and on the system architecture, the optimum number of keys per node must be obtained through experimentation. The experiment consists of a b-tree containing 10M keys. The population phase consists of inserting 10M random keys in such a way that every level but the last one is full. The last level of the tree is filled from left to right and, as a result, there will be some missing nodes in case the level is not completely full. This is the best case for the remote swap technique as some search operations will end in the penultimate level and one access to a different node will be saved. Figure 9 shows the execution time for a single search (average time over 500000 random number searches) when increasing the number of children in a b-tree populated with 10M elements.

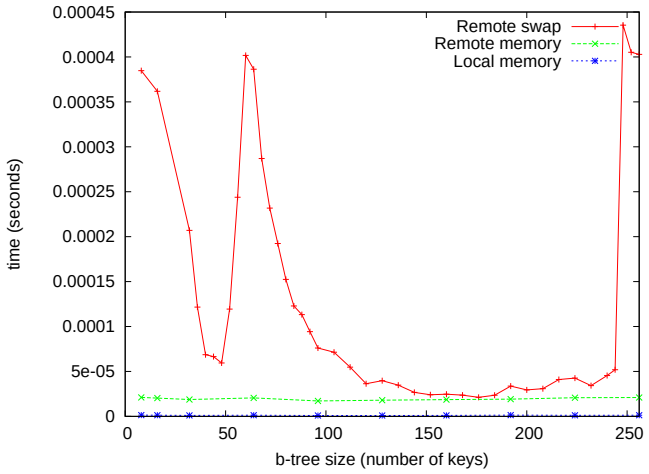


Fig. 9. Optimum number of children

As can be seen, the number of children has a great impact on the performance of the search operation in the remote swap scenario. The execution time varies depending on the number of keys that each node contains or, in other words, the number of levels in the tree that the algorithm has to inspect, typically equal to the number of new memory pages it has to visit. But there are more variables in the equation: for example, a particular alignment of the keys in memory can produce that the key array of a node is allocated in between two pages, what will produce a higher number of swaps. Also the position of the pointers to the children determines the number of swaps. As this can become a very complex study and our main objective is remote memory (not remote swap), we will not go through this analysis in depth. As shown in Figure 9, 168 seems a good approximation for the optimum number of children. From this number, we study the scalability of our system and remote swap technique. Figure 10 shows the results for this experiment.

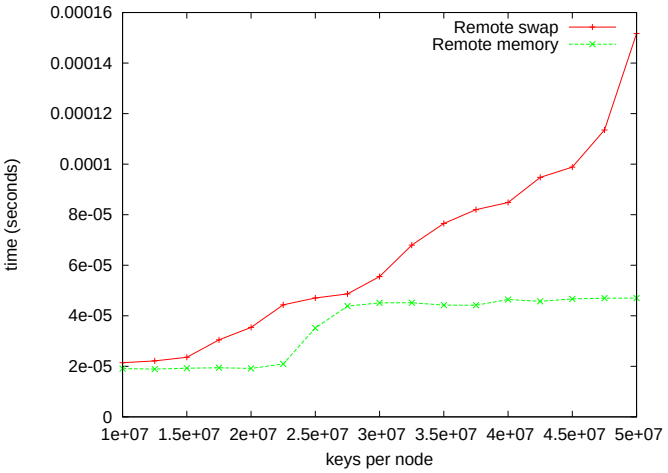


Fig. 10. Scalability analysis of data retrieval operations

The time required by the search operation increases with the number of keys present in the b-tree (the average depth), as expected. In the remote memory scenario, where the remote access has a constant latency independent of the page locality, the execution time increases in a linear way (there is a step in the graph due to minor differences in the implementation of the binary search in the node array and the search through the tree). On the contrary, the performance of the swap technique worsens exponentially, due to the page trashing syndrome.

C. Testing the Prototype with General Purpose Applications

Finally, Figure 11 presents the execution times for a set of PARSEC benchmarks [30]. We have chosen the benchmarks according to their memory footprint. Although the footprint is not important when testing remote memory (as we have previously said, we allocate remote memory explicitly), it is important to stress remote swap (the memory required by the application has to exceed the available local memory). In this way, benchmarks *blackscholes* and *raytrace* work satisfactorily with our prototype while the use of remote swap decreases their performance by a factor of two. When the memory footprint is quite large such in the case of *canneal*, the performance of remote swap worsens exponentially to prohibitive levels. Although there is a noticeable difference in the performance of our prototype and local memory for the *canneal* case, its execution is still feasible. Finally, regarding *streamcluster*, the memory footprint of this benchmark is small enough to fit in the local memory of the remote swap scenario, so no swap is needed.

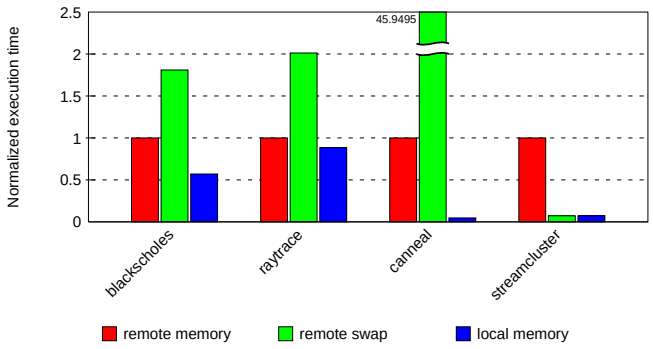


Fig. 11. Execution results for PARSEC benchmarks

VI. Conclusions

In this paper we have presented a first prototype that implements our proposal for non-coherent shared memory across a cluster. Although our prototype presents worse performance numbers than local memory in most of the scenarios, we are confident that improved implementations (the use of ASIC technology instead of FPGA) and the use of prefetching techniques will bring the performance closer to local memory. Nevertheless, our proposal allows the execution of applications that require large amounts of memory, which could only be executed in expensive mainframes otherwise.

Regarding the restriction in the amount of cores devoted to an application to those available in a single motherboard, we expect that this will not significantly limit the usefulness of our proposal. Given the current (and also near to mid-term) trends in processor development, motherboard implementations, and parallel programming, our proposal seems very promising as 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.

Our short-term objective is to continue testing the prototype with real applications or even databases. In this paper we have outlined a first incursion in databases through the search operation in a b-tree, but we aim to stress our prototype with a real full implementation, store indexes or the entire database in memory, and then study the execution time for different queries.

Finally, notice that all this study is directed towards proving that there are many cases where coherency is maintained by default, even if it is not needed, for example if the application has read-only phases. This way, the purpose of the system described in this paper is to provide an architecture that gets rid of the coherency overhead for memory-hungry applications that make use of the memory resources distributed across a cluster. This should be a key feature when developing scalable large shared-memory clusters.

References

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

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

[3] A. Acharya and S. Setia, "Availability and Utility of Idle Memory in Workstation Clusters," *SIGMETRICS Perform. Eval. Rev.*, vol. 27, no. 1, pp. 35–46, 1999.

[4] "Gaussian 03," http://www.gaussian.com, Gaussian 03.

[5] J. Gray, D. T. Liu, M. Nieto-Santisteban *et al.*, "Scientific Data Management in the Coming Decade," *SIGMOD Rec.*, vol. 34, no. 4, pp. 34–41, 2005.

[6] "3leaf Systems," http://www.3leafsystems.com, 3leaf Systems.

[7] M. Oguchi and M. Kitsuregawa, "Using Available Remote Memory Dynamically for Parallel Data Mining Application on ATM-connected PC Cluster," in *IPDPS 2000. Proceedings. 14th International*, 2000, pp. 411–420.

[8] J. Oleszkiewicz, L. Xiao, and Y. Liu, "Parallel Network RAM: Effectively Utilizing Global Cluster Memory for Large Data-Intensive Parallel Programs," in *Parallel Processing, 2004. ICPP 2004. International Conference on*, Aug. 2004, pp. 353–360 vol.1.

[9] T. Anderson, D. Culler, and D. Patterson, "A case for NOW (Networks of Workstations)," *Micro, IEEE*, vol. 15, no. 1, pp. 54–64, Feb 1995.

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

[11] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 267–278, 2009.

[12] F. Douglis, "The compression cache: Using online compression to extend physical memory," 1993, pp. 519 –529.

[13] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 74–85.

[14] Virident, "Virident's GreenGateway Technology and Spansion Eco-RAM," http://www.virident.com/solutions.php.

[15] Texas Memory Systems, "TMS RamSan-440 Details," http://www.superssd.com/products/ramsan-440/ .

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

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

[18] "Dynamic Logical Partitioning. White Paper," http://www.ibm.com/systems/p/hardware/whitepapers/dlpar.html, IBM.

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

[20] C. Keltcher, K. McGrath, A. Ahmed, and P. Conway, "The AMD Opteron Processor for Multiprocessor Servers," *Micro, IEEE*, vol. 23, no. 2, pp. 66–76, March-April 2003.

[21] J. Duato, F. Silla, S. Yalamanchili *et al.*, "Extending HyperTransport Protocol for Improved Scalability," *First International Workshop on HyperTransport Research and Applications*, 2009.

[22] H. Litz, H. Fröening, M. Nuessle, and U. Brüening, "A HyperTransport Network Interface Controller for Ultra-low Latency Message Transfers," *HyperTransport Consortium White Paper*, 2007.

[23] H. Fröening and H. Litz, "Efficient Hardware Support for the Partitioned Global Address Space," in *10th Workshop on Communication Architecture for Clusters*, April 2010.

[24] H. Fröening, M. Nuessle, D. Slogsnat, H. Litz, and U. Brüening, "The HTX-Board: A Rapid Prototyping Station," in *3rd annual FPGAworld Conference*, Nov. 2006.

[25] D. Slogsnat, A. Giese, M. Nüssle, and U. Brüning, "An Open-source HyperTransport Core," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 1, no. 3, pp. 1–21, 2008.

[26] H. Midorikawa, M. Kurokawa, R. Himeno, and M. Sato, "Dlm: A distributed large memory system using remote memory swapping over cluster nodes," 29 2008-oct. 1 2008, pp. 268 –273.

[27] S. Liang, R. Noronha, and D. Panda, "Swapping to remote memory over infiniband: An approach using a high performance network block device," sept. 2005, pp. 1 –10.

[28] P. Machek, " Network Block Device (TCP version)," http://nbd. sourceforge.net/.

[29] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes," pp. 245–262, 2002.

[30] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.