MEMSCALE^{TM}: a Scalable Environment for Databases

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

***Abstract*—In this paper we propose a new memory architecture for clusters referred to as MEMSCALE. This architecture provides a distributed non-coherent shared-memory view of the memory resources present in the cluster. With this aggregation technique, a given processor can directly access any memory address located at other nodes in the cluster and, therefore, the whole memory present in the cluster can be granted to a single application.**

In this study we focus on in-memory databases as a memory-hungry application in order to show the possibilities of our new architecture. To prove the feasibility of our idea, a 16-node prototype cluster serves as a demonstrator. Part of the memory in each node is used to create a global memory pool of 128GB which hosts an entire database. First we show that providing more memory than usually available in a typical commodity node for a database server makes the execution of queries more than one order of magnitude faster than using regular SSD drives. After that, we go one step further and show that simultaneously accessing the database from all the nodes in the cluster converts our prototype into a powerful database server capable of beating current commercial solutions in terms of latency and throughput.

Keywords: memory architecture, cluster computer, non-coherent memory, in-memory databases

I. Introduction

Commodity computers have become the common building block for scalable high performance computing. As a matter of fact, 83% of the systems included in the Top500 list are cataloged as cluster computers [1]. The main reason is that clusters based on commodity computers are noticeably much more cost-effective than their counterpart massive parallel processing systems.

However, the cluster architecture partitions the system memory into isolated pieces, each one located at a different node. In this way, communication among nodes is resolved by exchanging messages, although this access to foreign memory undergoes extra overhead caused by the message handling layer (in addition to the higher latency due to the distance between nodes). Nevertheless, this paradigm is commonly used by MPI-based applications.

This innate latency in the process of gaining access to remote memory through a message, together with the required extra effort when dealing with explicit messages on the programmer side, encourages the use of shared-memory applications where possible. However, as a processor can only directly access memory allocated at its node, the habitat of a shared-memory application is restricted to a single motherboard, thus hindering its use across a cluster. However, the current trend in the number of cores per socket alleviates the previous restriction in the sense of computing power resources: nowadays, it is easy to configure a motherboard with 32 cores and, as this number will increase up to 80 cores in the near term, the number of execution flows hosted in a single node can be quite high. However, note that many shared-memory applications do not scale beyond a few tens of threads [2], either because of synchronization problems or because unbalances in the system such as I/O bottlenecks in some data-intensive applications.

But this situation changes with regard to memory needs, which are a harder requirement than the computing power one: a decrease in the number of available cores produces a linear increase in execution time, but a decrease in the amount of available memory produces an exponential increase in execution time. This behavior is due to the fact that secondary memory storage makes up for the lack of main memory, although their performance differs in several orders of magnitude. This is why memory is overscaled at each node in clusters, just to prevent the critical situation where an application runs out of main memory. However, most of the time this just-in-case memory remains idle (but consuming power). This economic cost and energy inefficiency is not the only problem. As described in [3], current trends in DIMM technology predict that the amount of available memory per core will drop by 30% every two years. This means that applications will become more and more memory restricted and, thus, a remedy for the memory *capacity* wall seems to be urgent.

We proposed a solution in [4][5] to increase the available memory to an application by leveraging main memory from the other nodes in a cluster. As we explain later, this approach, called MEMSCALE^{TM}, can be seen as a memory aggregation mechanism that does not require coherency among nodes in the cluster because the global memory pool is treated as an exclusive distributed memory, that is, only one application located in one of the nodes can use this memory at a time. In this paper we apply our remote memory architecture to databases and analyze how this kind of applications can benefit from a large main memory pool.

By nature, databases present an insatiable need for memory. Due to the large amount of data that these applications usually handle, tables have been traditionally stored in secondary memories like hard disks. However, as the

latency of accessing main memory is orders of magnitude faster than accessing hard disks, interest on in-memory databases has grown as a way to increase performance. We can find many examples of commercial implementations of this idea, for example IBM solidDB [6], Oracle TimesTen [7] and McObject [8] just to mention some of them. Also Google moved their indexes from disk to main memory in order to increase in throughput and decrease in latency [9]. Additionally, a system specially designed for databases is the Oracle Exadata Database Machine [10], that provides two database servers (among other I/O servers), each one with 64 cores and 1TB of main memory. The proportion of memory and computing power is 16GB per core, a quite high rate compared to commodity computers. As can be seen, databases are highly coupled to main memory availability, especially for queries with critical time restrictions.

The remainder of this paper is structured as follows: we show related work in the next section, followed by a description of our MEMSCALE architecture in Section III. Section IV focuses on our MySQL implementation and Section V describes the case study, which is evaluated in Section VI. Later, in Section VII we present and evaluate the actual proposal in this paper: an extension to our MEMSCALE architecture. The last section draws some conclusions.

II. Related Work

Recent advances in solid-state memory technologies, particularly NAND Flash memories, have allowed the introduction into the market of new storage systems based on these non-volatile components. Thus, we can find a range of products intended to increase I/O performance by leveraging solid state drive (SSD) lower latency and higher throughput (especially at reading) [11]. Together with the emerging of this technology, a new trend from the application side makes even more attractive this kind of storage: in contrast to traditional data-intensive applications such as database analytics and datamining, whose access patterns have become more and more disk-friendly (sequential accesses, bulk reads, etc.), a new kind of data-intensive applications with different access patterns and performance requirements are becoming more common in current datacenters. This new breed of applications is directly connected to services available in Internet like social networks such as Facebook, global searchers like those provided by Google, messaging applications like Hotmail, etc. The common characteristic among these Internet services is that data is intensively accessed by means of multiple short transactions. This fine grain access pattern requires a storage technology that presents good performance even with random accesses, and this is why SSD storage is becoming popular in these contexts. In academia we can also find studies about SSD storage and its applicability to databases [12].

With regard to random access, even better than SSD is, of course, RAM memory. The previous mentioned McObject solution reports an interesting experience: 1TB database is loaded into main memory by leveraging an SGI Altix 4700 server [13] that provides a single and unified system image. However, due to the fact that coherency is maintained throughout the system, the scalability and performance of this approach are limited. The main difference between the SGI Altix and MEMSCALE is the additional overhead of the global coherency protocol in the case of the Altix, as our approach does not keep coherency across the nodes of the cluster. Other commercial solutions like Numascale [14] or ScaleMP [15] also provide coherency and undergo this overhead.

The solutions by SGI, Numascale, or ScaleMP can be classified as vertical scaling solutions, that is, they increase database performance by improving the underlying hardware infrastructure in order to provide one single, but much larger, system image. On the contrary, other solutions follow a horizontal scaling approach. In this case, commodity computers are attached to the existing infrastructure in order to achieve a smooth scaling in a distributed fashion by creating a loosely-coupled architecture with better potential scalability. For example, we can focus on MySQL Cluster [16], one of the most popular database engines for computer clusters. Basically, MySQL Cluster uses data partitioning and allocates each portion of the database in the main memory of each node of the cluster. Nodes communicate among each other over traditional interconnection networks, like Ethernet or Infiniband for instance, through a socket interface.

III. The MEMSCALE™ Architecture

Although databases are widely discussed all along the paper, it is important to remember that this study is about memory architecture. In this way, databases are just the application we use in order to demonstrate the characteristics of our proposal. In this section we present the insights of our system: MEMSCALE, which can be seen as a new distributed memory system for clusters. The key factor of our proposal is the way in which processors access memory physically located at other nodes. Two ideas have been present while developing this architecture:

- Achieve the lowest possible latency for remote accesses
- and do it without harming the ease of programming of shared memory.

In order to adhere to the second requirement, any shared memory application should be successfully executed in our system without requiring its code to be modified. Actually, even a preexisting binary can be executed in our architecture. If so, the question is how remote memory accesses are triggered. In a software distributed shared memory architecture (DSM) or in a remote swap system, the event that triggers the remote access mechanism is the page fault [17], which presents high associated overheads like OS traps and page-level data migration. On the contrary, the trigger for accessing remote memory in our system does not exist because accesses to remote memory are not treated in any special manner. In this way, a remote read or write is actually a regular load or store assembler instruction and, therefore, it does not need any dedicated handling by the processor or operating system. Of course, there is one page fault at the

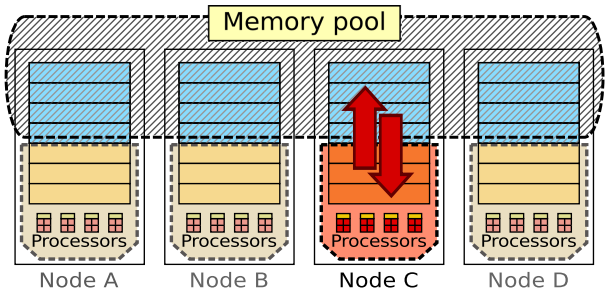


Figure 1. Example of a global memory pool created from four nodes and accessed from one single node

beginning of the execution but, after that, accesses to remote memory will not generate a fault, as remote memory is part of the processor memory map. As a result, latency to remote memory is significantly lower as accessing remote locations is entirely done by hardware thus additionally adhering to the first requirement above. However, note that the main difference is not related to latency, but to the nature of the architecture, as our system is not *event-based* and does not involve any software support for accessing remote memory.

The lack of coherency across the nodes of the cluster is the second reason for our low latency to remote memory. Traditionally, the overhead of the coherency protocol has been the limiting factor for large shared-memory computers: keeping coherency among such a high number of processors becomes prohibitive as memory operations turn out to be too expensive in terms of time. Briefly, the function of a coherency protocol is that when a write operation finishes, the new data value becomes visible to every processor in the system. This restriction jeopardizes the scalability for large systems in the presence of caches as old data may be stored in them and, therefore, an invalidation (or update) process must take place before the write operation can be committed. In order to avoid these constraints and therefore keep remote memory access time as low as possible, our system does not maintain coherency among nodes in the cluster. As a result, our system can be seen as a group of coherency domains, each one constraint to a single node as our architecture does not change the original behavior of mainstream processors, thus allowing them to remain coherent within a single motherboard. However, there is no such thing as a free lunch: if caches from distinct nodes are not kept coherent, we may have troubles at executing an application spread through those nodes. In this way, at a first stage, let us focus on the scenario depicted in Figure 1.

In Figure 1 we see a global memory pool made up of some main memory portions from each node. The rest of main memory is granted to the corresponding node in private mode to be used by the operating system (note that each node has an independent OS) or other non-shared structures. In the example shown in Figure 1, only one node will access the memory pool. However, note that multiple memory pools can be simultaneously configured allowing multiple nodes to have access to distinct exclusive memory pools. Nevertheless, in such a multi-pool configuration, a given node only has access to the memory pool associated to it. Further details about the MEMSCALE architecture can be found in [4][5].

As can be seen, MEMSCALE allows the execution of memory-hungry applications in the cores present in a single node but using as much memory as needed, not limited to the memory physically attached to that node. In this case, there is no need for propagating invalidation messages to the other nodes because a given memory address will only be present in the caches contained in a single node. In this way, the lack of coherency among nodes in the cluster does not impose any limitation in this scenario. A more ambitious scenario will be presented in Section VII, which is the actual architectural proposal of this paper.

A. *MEMSCALE™ Implementation*

As we said, in order to access memory located in other nodes, our proposal does not require applications to be modified as accessing remote memory completely relies on hardware. Actually, applications do not need to be aware that they are making use of remote memory. In our approach, processors do not distinguish between local and remote accesses as both are the result of regular load or store instructions. If the addressed memory turns out to be remote, then the memory access will automatically be forwarded to the corresponding remote node. We accomplish this by means of HyperTransport, although other protocols such as QPI or PCIe could also be used.

HyperTransport (HT) [18] is used to interconnect the AMD Opteron processors in a motherboard, where each processor is attached to part of the physical memory by means of its own memory controller. Therefore, as there are several memory controllers, processors require to know where to forward a given memory request. This is achieved by including at each processor a set of address base/limit registers configured at boot time. Hence, when a load or store operation related to a given memory location is issued, the processor compares the requested address with those registers, and then the memory request is forwarded to the memory controller responsible for that memory address. Forwarding the memory operation involves the generation of a HyperTransport packet.

The system described above is the basis upon which we will implement our proposal, which involves creating a new hardware component that will be referred to as *Remote Memory Controller* (RMC) [19]. This new component will be presented to the processors in the motherboard as a HyperTransport I/O Unit and will be responsible for forwarding memory requests from the local system to remote nodes. After properly configuring the mentioned address base/limit registers, processors will automatically forward HT transactions to their local RMC, which will forward these requests to the corresponding remote node, where these transactions are handled by the memory controllers of the remote CPUs. Obviously, before accessing remote memory, a reservation phase that assigns remote memory to the process must be carried out. This reservation process is not covered in this paper because it is not relevant to

the experiments, its implementation is trivial, and it has no impact on the system performance (e.g. an in-advance reservation policy can easily relegate this mechanism to a minor detail).

B. MEMSCALE™ Prototype

We have built a prototype that implements our proposal for non-coherent distributed 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, accounting for a total of 256GB of RAM. Notice, however, that although this is the maximum amount of memory we can currently share in our prototype, nothing except the economical cost of memory prevents us to provide up to 2TB of memory to an application, as the used motherboards can hold up to 128GB (indeed, one extra node equipped with 128GB will be used in the tests to draw an optimistic upper bound).

The motherboards used include an HTX connector[1], where we have attached an add-in card [20]. This card includes an FPGA that will be used to implement the RMC functionality. It also includes six fiber links. We will use four of them to interconnect the 16 nodes in a 4x4 2D mesh. The routing functionality will also be implemented in the FPGA. The HT interface is running at 200MHz (HT400), although the remaining FPGA logic runs at 156MHz.

IV. Porting MySQL to MEMSCALE™

In this study we aim to analyze how much large-scale databases can benefit from our system. To do so, we chose MySQL 5.1 as it is one of the most popular databases. Moreover, its open source nature was also a key factor in our decision.

Although we previously claimed that any shared-memory application can be executed in our system without even recompiling, it is also possible to enhance the application code in order to have more control over which of the application data are allocated in local memory and which in remote memory. These minimal changes allow a thorough study of the application behavior by strictly controlling the amount of remote memory in use. In this section we explain how MySQL is adapted to our proposed architecture according to the scheme depicted in Figure 1.

MySQL is designed so that there is an abstraction layer, referred to as *storage engine*, between the server logic and the data storage. The goal of the storage engine is to provide MySQL with some degree of adaptiveness to the different existing storage architectures. Therefore the same server logic can interact with different types of storage by using a single well-defined interface.

There are several storage engines currently available. The storage engine named *memory* (also known as *heap*) is particularly interesting. As its name suggests this storage engine stores data in main memory. It is typically used for creating temporary tables that can be accessed at a very high speed. We leveraged in MEMSCALE this storage engine, as most of the work was already done. To illustrate the easiness of use of our system, we just took the source code of the heap storage engine and substituted the memory allocation functions (*malloc* and related ones) with our remote memory allocation functions. As the rest of the code does not care if an array is stored in local or in remote memory, we only had to compile the storage engine. At this point, we were able to execute any SQL statement on tables stored in remote memory.

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

V. Performance Evaluation: the Case Study

For this study we have designed a test bed consisting of a large database structure and a set of queries. Although there already exist some database benchmarks, they do not fulfill our needs for this particular study. For example, TPC-H from the Transaction Processing Performance Council comprises a set of decision support queries, mainly retrieval queries (read-only) with a high degree of complexity. However, in this paper we aim to recreate a kind of database load in a much more dynamic fashion, in particular with a high number of concurrent short queries. Actually, our intention is to mimic a typical social network design. Another popular benchmark is TPC-C, that emulates an OLTP (on-line transaction process) database load. However, in this case some of the queries in TPC-C are write queries. However, in this study we do not target this kind of queries that modify the state of the database because of the fact that read-only queries are such a common database usage that deserves this dedicated study and because the meaning of write latency depends on the consistency model of the database, topic that is out of the scope of this paper.

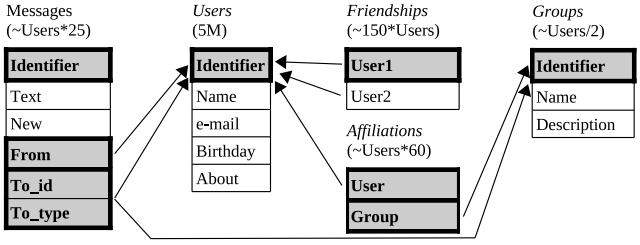


Figure 2. Table structure (gray background indicates an indexed field)

In order to represent the scenario intended for this paper, we have created the database structure depicted in Figure 2. The size for this database is around 100GB including indexes. Additionally, we have designed six short retrieval queries. We define a short query as a query that walks through tables exclusively by using indexed fields and that retrieves no more than a few hundred records. This allows an execution time below one second for a single query even in this large database. Query descriptions are presented below.

- Retrieve the name of the friends of a given user.
- Retrieve the number of new messages sent to a given user.

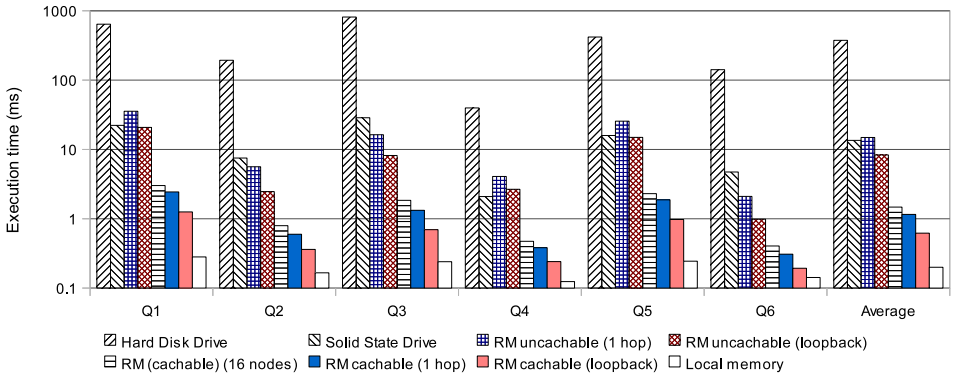


Figure 3. Comparative analysis of query execution time

- Retrieve the number of new messages sent to the groups that a given user is affiliated to.
- Retrieve last 100 messages sent by a given user.
- Retrieve the text of a given message.
- Retrieve the name of the groups that a given user is affiliated to.

VI. Performance Results

In this section we compare the performance achieved by a database leveraging MEMSCALE with the performance of other database storages, like HDD, SSD, or in-memory databases leveraging regular RAM memory local to the node accessing the database.

As a typical configuration, the MySQL server runs in a single node (queries are executed in the processors of a single motherboard). In this way, memory storages can be compared against the usage of traditional storage engines (MyISAM) based on HDD or SSD drives. Therefore, the server can potentially be scaled up to the number of cores present in the motherboard. Moreover, MySQL server can also scale by using main memory as a data cache level between secondary memory and the cores. In order to test these scenarios, the node has been configured with 8GB of main memory. This memory will be used to cache parts of the database mainly in two ways: actively by the database server or indirectly by the operating system. Anyway, caching will not play an important role due to the random access nature of the queries. Additionally, the configuration for SSD is composed of two Kingston SNV425-S2 64GB drives configured in RAID 0 for improved performance, each of them with a sequential speed of 200MB/s at reading and 110MB/s at writing. The configuration for HDD is: Seagate Barracuda SATA 3Gb/s, 32MB cache, 7200RPM and average latency of 4.16ms.

On the other hand, we also test the original *memory* storage engine, that will use regular local main memory as a pure storage space. In this case, the node will be configured with 128GB of main memory, so that the entire database could be loaded in this memory. This node configuration represents an utopian configuration, as it is only valid for medium-size databases due to the limited amount of RAM memory that can be attached to a motherboard. Nevertheless, this configuration will serve as an upper bound for our MEMSCALE system. Finally, the *remote memory* storage engine enables the server to scale up by using memory from other nodes. As described in Figure 1, queries are executed in one node against a database stored in the memory pool spread over the cluster (16 nodes in this study). Additionally, in the present section, we will focus on a more controlled environment consisting of one node (8GB main memory) that executes queries while the entire database is stored in another node (128GB main memory). Note that this is not the intended configuration for our architecture, but in this section it comes in handy in order to isolate remote memory latency.

In Figure 3 we show the different execution times for the storage engines under study. The execution time refers to the time spent by a single query (therefore, this is an average over thousands of queries). Note that time axis is in logarithmic scale and also that queries have been executed sequentially, that is, only one query flow (this is the best achievable time).

We can see that the execution time proportion for the different scenarios and between the used storage engines has no significant differences among the query types. Minor variations in these proportions are mainly due to different access localities (note that HDD and SSD benefit from memory page reutilization, that is, sequential access). Attending to average times, the first conclusion is that SSD is 28 times faster than HDD, because SSD technology has a better random access latency (no need for moving parts, header, or disk spinning). However, local main memory is 65 times faster than SSD, due to the fact that RAM memory presents better latency and higher bandwidth, but also because there is no need for accessing a secondary storage, so the operating system is not involved in terms of I/O handlers. However, the amount of memory present in a single node has limitations, either economic or technical, and this is the rationale for using our remote memory system for large databases.

Regarding our proposal, the latency to remote memory

depends on the distance (hops from source to destination) between the core accessing remote memory and that memory host. In Figure 3, we take as a reference the case for one-hop distance. We can see that our system, when caches are enabled, is more than one order of magnitude faster than SSD, and local memory is only six times faster than our approach. However, note that our prototype is currently based on an FPGA implementation and latency will decrease when using an ASIC. To show the overhead in the access to remote memory due to the FPGA implementation, we have introduced numbers for loopback mode, that is, processors access their own local memory but through the FPGA. The difference in execution time between loopback and local memory illustrates part of the mentioned overhead. Additionally, for a more comprehensive comparison, we have also included a scenario in which the data is spread across 16 nodes. It can be seen that this case only differs on slightly higher average remote memory latencies with minimal performance impact. Finally, we can see results for remote memory when caches are disabled. The importance of caches here is not due to data caching from previous queries but to cache data being used inside a single query. Note that when caches are disabled, the used processors limit each remote load to a maximum of 64 bits, while for enabled caches each remote load retrieves 64 bytes (i.e., one cache line).

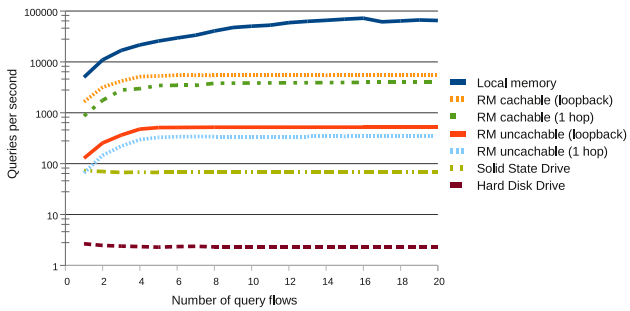
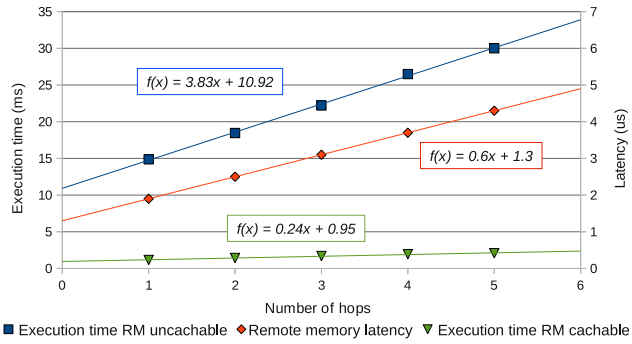


Figure 4. Maximum throughput in various scenarios

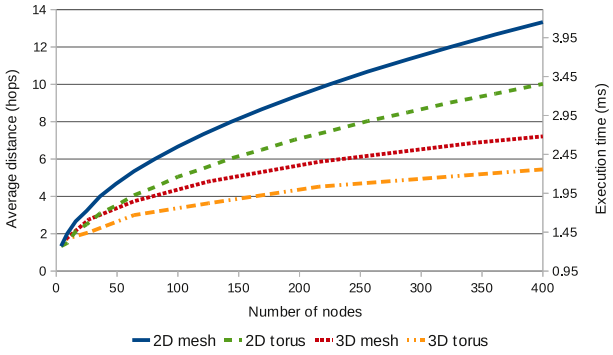
Figure 4 shows results for our second study in this section. In this case, we analyze the scalability possibilities when the server is fed with a higher number of concurrent queries. To do so, instead of using only one query flow, we have executed different threads that send to the server queries in parallel. The target of this test is to analyze the maximum throughput of the different storage engines.

Aside from the performance differences previously seen, let us focus on how the number of queries per second increases with the number of parallel queries sent to the server. We can see that the *memory* engine perfectly scales (note the logarithmic scale) up to 16 threads, that is, the number of cores in the motherboard. On the contrary, both HDD and SSD do not scale even to two cores, because I/O quickly becomes the bottleneck, not the processors. In this way, we can see that the maximum throughput does not increase when increasing server load. However, the *remote memory* engine scales up to four threads. This limit is imposed by the current configuration of the Remote Memory Controller, which is currently designed as an I/O device. In this way, the maximum number of outstanding requests to remote memory is limited to one per socket. The solution to overcome this limitation would be to implement the RMC as a regular memory controller, i.e. to participate in the local coherency protocol.

As mentioned before, the distance between cores and remote memory is crucial to the system performance. In Figure 5(a) we see some experimental results when incrementing distance and their corresponding equations generated by linear regression. In order to run these experiments, we have increased the number of nodes in between the data node (128GB) and the node where queries are executed. Just to understand the relationship between latency and execution time, note that when moving from one hop to two hops, latency increases by 32%, while the execution time increases by 26% without caching and by 20% with caching. In Figure 5(b) we extrapolate these results to predict the execution time when the number of nodes increases (caches enabled). For example, with 256 nodes in a 2D mesh (16x16), execution time increases by 95% compared to two nodes, and when using 216 nodes in a 3D mesh (6x6x6), execution time is 27% higher than using two nodes. Note that execution times in Figure 3 for 16 nodes are in-line with these predictions. This, in particular, shows the feasibility of our approach in terms of scalability.



(a) Relation between remote memory latency and query execution time



(b) Predicted query execution time for various topologies

Figure 5. Analysis of MEMSCALE scalability

VII. A New Degree of Performance in MEMSCALE™

Up to this point, we have seen how a MySQL server running in one node is able to store and load data from any memory location in the whole cluster. As seen in the previous section, for large databases that do not fit into regular local memory, this feature improves database response time per se. In this section, we go one step further and introduce an evolved system where not only memory is aggregated but also processors are, converting the entire cluster into a powerful database server. In this way, while in the previous section only the processors in a single node could hold the execution of queries, now any query can be executed in any node of the cluster. As can be seen, the potential performance achieved by the cluster has gained a new dimension. If response time for a single query was improved in previous section, now the number of queries served in parallel is increased as well.

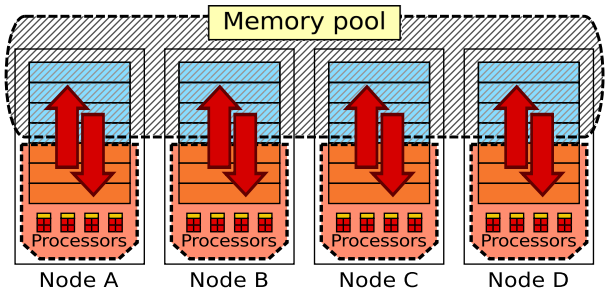


Figure 6. Example of a global memory pool created from four nodes and accessed from every node

Figure 6 presents the cluster configuration for this proposal, where every node in the cluster has concurrent access to the global memory pool. In this way, we not only make the most from the memory present in it but also from its computing power. To achieve this target, each node will host a MySQL server. The key idea is to share the required data structures among the servers so that the cluster could behave like a single large server with multiple entry points.

In order to port MySQL to this multi-node configuration, we must notice that table rows are already stored in the global memory pool, but they were previously accessed in a exclusive way by only one server. However, as structures containing table information are stored in the memory pool too, each individual server in the multi-node configuration will automatically see the global data at startup. As can be seen, porting MySQL to our new architecture is extremely easy to accomplish.

Finally, due to the lack of coherency among nodes in our proposed architecture, concurrent writes and reads may produce an erroneous result because a query may read a stale value from a non-invalidated cache. However, this issue can be managed by means of explicit cache invalidations (anyway, note that as coherency is maintained inside each node, the single-node configuration described in the previous section supports write queries indeed). Nevertheless, this matter is not elaborated in this study as explained in Section V.

A. Performance Results

For these experiments, a 128GB memory pool is created from 16 nodes. In this scenario, each node shares 8GB for the memory pool and keeps 8GB for private use. Figure 7 shows the results for these experiments. We can see that our 16-node prototype system scales up to 80 concurrent query flows. Note that, as seen in the previous section, a single node scales up to 4 or 5 query flows. In this way, 80 flows in a 16-node cluster seems to be a linear scalability. Up to this number of concurrent flows, the response time remains quite low and, once arrived to saturation, this response time starts to increase to unacceptable times (response time is normalized to the value of one query flow). With 80 concurrent query flows we achieve a throughput of 35000 queries per second. This throughput will presumably increase when the number of outstanding requests per socket also increases. In this case, scalability will also increase, being linear with the number of cores instead of linear with the number of sockets. Finally, note that a bigger cluster will produce a higher throughput with little penalty, as predicted in Figure 5(b).

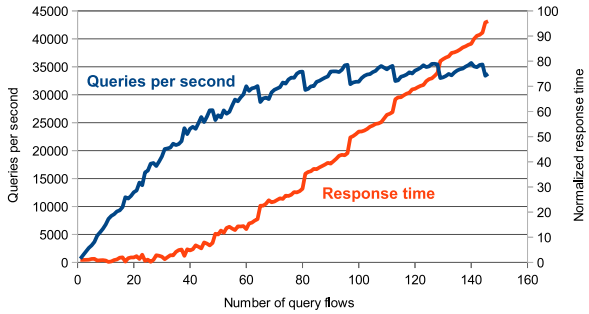


Figure 7. Results for the multi-node server configuration

B. Comparing to MySQL Cluster

In this second part of the study we have evolved our system so that the entire cluster can act as a single database server. In this way, we can no more compare with standard MySQL because MySQL will only use a single node. However, there is a related system, MySQL Cluster, that allows the use of a cluster to create a single large database server that stores data in main memory too.

However, MySQL Cluster has some limitations. Its reference manual recommends that all the records accessed by a transaction must be held in and serviced by the same node [21]. This means that when the process of executing a query needs data stored in different nodes, a noticeable overhead should be expected due to the retrieval operation.

To illustrate the problem of these queries in MySQL Cluster we have reran the throughput experiments. Note that, in this case, nodes in our cluster use a regular 1Gbit

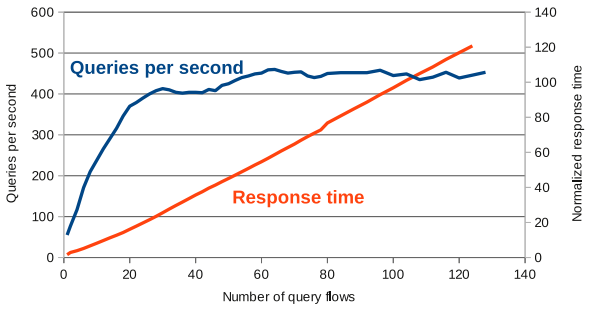


Figure 8. Results for MySQL Cluster

Ethernet interconnect. Figure 8 summarizes the results. Average latency for a single query is 18ms, slightly higher than SSD. However, its throughput is near seven times better than SSD, due to the distributed nature of MySQL Cluster. But compared to our system, MEMSCALE has a throughput about 77 times higher than MySQL Cluster. Although there is probably room for improvement on the configuration and tuning of MySQL Cluster and a better interconnection may reduce network latency, the throughput difference near two orders of magnitude shows that MEMSCALE is clearly more powerful.

VIII. Conclusions

In this paper we presented a new memory architecture for clusters and tested how well this system suits large databases. As our proposal presents to processors memory in other nodes not as a secondary or a special level in the memory hierarchy but as a continuous expansion of its own main memory, access to that remote memory can be achieved by a regular read or store operation with very low latency. In this way, we saw that queries that present high I/O requirements have an execution time one order of magnitude lower in our system than in a system that relies on HDD or SSD drives. Regarding concurrency, HDD and SSD drives act as a bottleneck to the system while our proposal is able to manage a higher number of concurrent queries. From a scalability point of view, we can leverage all available computational resources in the system in order to dramatically increase the query throughput. At this point, the number of queries that the server can handle at a time grows linearly with the number of nodes in the cluster without seriously harming their response time. Although our prototype uses FPGA technology, it achieves a notable performance, even better than commercial solutions like MySQL Cluster.

Acknowledgments

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

References

[1] "Top 500," http://www.top500.org.

[2] 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.

[3] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *Proceedings of the 36th annual international symposium on Computer architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009, pp. 267–278. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555789

[4] H. Montaner, F. Silla, and J. Duato, "A practical way to extend shared memory support beyond a motherboard at low cost," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 155–166. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851495

[5] H. Montaner, F. Silla, H. Fröning, and J. Duato, "Getting rid of coherency overhead for memory-hungry applications," in *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, 2010, pp. 48 –57.

[6] "SolidDB: In-Memory, relational database software for extreme speed." http://www.ibm.com/software/data/soliddb.

[7] "Oracle TimesTen In-Memory Database," http://www.oracle.com/us/products/database/timesten.

[8] "In-Memory Database Systems: Pushing Past the Terabyte-Plus Boundary," White Paper: http://www.mcobject.com/terabyte-plus-benchmark.

[9] Jeff Dean, "Challenges in Building Large-Scale Information Retrieval Systems," http://research.google.com/people/jeff.

[10] "Oracle Exadata Database Machine," http://www.oracle.com/us/products/database/database-machine.

[11] "Fusion-IO," http://www.fusionio.com.

[12] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, "A case for flash memory SSD in enterprise database applications," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 1075–1086. [Online]. Available: http://doi.acm.org/10.1145/1376616.1376723

[13] "High Performance Computing: SGI Altix 4700," http://www.sgi.com/products/servers/altix/4000, SGI.

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

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

[16] "MySQL Cluster," http://www.mysql.com/products/cluster.

[17] M. Chapman and G. Heiser, "vnuma: a virtual shared-memory multiprocessor," in *Proceedings of the 2009 conference on USENIX Annual technical conference*, ser. USENIX'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 2–2. [Online]. Available: http://portal.acm.org/citation.cfm?id=1855807.1855809

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

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

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

[21] Oracle, "White Paper: MySQL Cluster Evaluation Guide - Designing, Evaluating and Benchmarking MySQL Cluster," http://www.mysql.com/products/cluster/resources.html.