MEMSCALE: In-Cluster-Memory Databases

Live Demo

Héctor Montaner
Uni. Politècnica de València
Camí de Vera, s/n 46022
València, Spain
hmontaner@gap.upv.es

Federico Silla
Uni. Politècnica de València
Camí de Vera, s/n 46022
València, Spain
fsilla@disca.upv.es

Holger Fröning
University of Heidelberg
B6, 26, Building B (3rd floor)
68131 Mannheim, Germany
froening@uni-hd.de

José Duato
Uni. Politècnica de València
Camí de Vera, s/n 46022
València, Spain
jduato@disca.upv.es

ABSTRACT

We have developed a new memory architecture for clusters that allows automatic access from any processor to any memory module in the cluster completely by hardware. Thus, with a single assembly instruction a processor can retrieve (or update) a memory location in a remote node. The efficiency of this new paradigm makes it possible to speed-up the execution of shared-memory applications with very large memory footprints by running them across the entire cluster, thus providing them a true shared-memory environment (contrary to the emulation typically carried out by software-based distributed shared memory).

This new memory architecture, referred to as MEMSCALE, opens up a new frontier for memory-hungry applications. In this paper we focus on in-memory databases and show how this target application can be boosted by our memory architecture, which can virtually provide unlimited memory resources to it.

In the demo presented in this paper we show the advantages of our architecture by means of a prototype cluster. We configure two cluster sizes, 16 and 32 nodes, to analyze throughput scalability and latency worsening, to extrapolate these metrics to bigger clusters, and to show the benefits of our technology compared to other alternatives like SSD-based databases. Moreover, we also show the easiness of use of our architecture by explaining how we ported MySQL Server to our prototype cluster. Finally, the possibility of executing queries in any processor of the cluster during the live demo will show the audience how our system aggregates the advantages of the scale out and scale up approaches for database server growing.

1. INTRODUCTION

Database software has traditionally relied on secondary storages, pushing into the background the role played by main memory. This was mainly due to the size of databases, usually much larger than the amount of RAM memory available in the computer. However, as the amount of main memory in commodity computers has been continuously increasing to considerable quantities, the caching technique has gained popularity. For example, Memcached[7] is a system for caching objects into main memory aimed to scenarios where a minor percentage of data is targeted by a major percentage of queries.

Although Memcached has substantially improved performance in many case studies, this is just a conservative extension of the old-fashion secondary storage databases. What brings a quantum leap to databases is *in-memory* databases. These databases store data natively in main memory and benefit from its random access characteristic. Thus, contrary to caching systems that only allow access to a cached object in a key/value fashion, an in-memory database allows the execution of SQL queries directly against tables stored in main memory. This new paradigm has many advantages, for example the use of hash tables instead of *b-tree* structures. This shift in data storage noticeably improves the performance of queries that present a random data access pattern.

Placing the database data in main memory is not the only factor that improves performance of in-memory databases: *concurrency* also matters. In this regard, with previous secondary storage, concurrency was highly constricted by hard disks as they acted as bottlenecks and one single query flow could easily saturate the disk throughput. But when talking about RAM memories two characteristics lead us to expect higher performance: on one hand, regular multicore commodity computers are designed so that all the cores can access memory in parallel without reaching saturation and, on the other hand, as main memory is accessed via hardware, there is no potential bottleneck in software handlers. In this way, in-memory databases should be a synonym for concurrency. Therefore, the current trend towards increasing the

number of cores per processor does nothing but emphasize the need for in-memory databases to the detriment of classic secondary storages.

However, despite the growing popularity of these in-memory databases (Oracle TimesTen[3], IBM SolidDB[5], etc.), the limited amount of memory in a single node is clearly the main constraining factor for their widespread applicability: although nowadays we can find commodity computers with 512GB of main memory, it is not enough to host many mid-size databases, or not even their indexes. In order to efficiently increase the amount of memory available to a given application, we have devised a new non-coherent distributed shared-memory architecture for clusters referred to as MEMSCALE[1].

In this paper we present the use of this new memory architecture for in-memory databases. Our proposal attacks this scenario by reducing latency and increasing throughput, in terms of queries per second, to achieve a highly scalable database system. The rest of the paper is organized as follows: in next section our memory architecture is briefly described. In Section 3 we introduce the most popular related work and some performance comparisons are outlined. Finally, in Section 4 we describe the live demo to be presented.

2. THE MEMSCALE ARCHITECTURE

According to the TOP500 list, 82% of the most powerful computers in the world are cataloged as clusters. The reason for this is twofold: the good scalability that this distributed architecture presents and the high ratio performance per dollar of commodity computers. Therefore, it seems reasonable to base MEMSCALE on a cluster of computers. The objective of our proposal is to convert the entire cluster into a single database server that is the result of aggregating all the processors and main memory in the cluster in order to reach previously unseen performance.

Figure 1 summarizes the MEMSCALE architecture. It can be seen that a memory pool is created out of the individual memories of each node in order to configure a shared memory region in the cluster accessible by any processor. Tables and indexes of the database will be loaded into this memory pool. Notice that tables may become partitioned across several nodes as each node will host a portion of registers in its main memory. However, the resulting effective structure is not a partitioned database because all the nodes in the cluster see a continuous memory address space that they can access independently of which node is hosting the requested memory location. Therefore, spatial allocation of data is not relevant in practice to processors as they see a global memory pool working as a big shared-memory mainframe with lots of memory.

The key factor in our system is the remote memory direct access feature. To achieve this, we have leveraged Hypertransport[6]. Hypertransport is the technology used to interconnect the AMD Opteron processors present in a single motherboard. Our implementation extends this interconnection to allow direct communication among motherboards. To do so, we have developed a new hardware component that we will refer to as *Remote Memory Controller* (RMC) and that is implemented in the card shown in Figure 2, which uses the

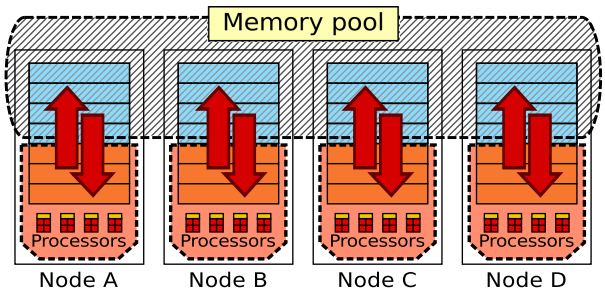

Figure 1: Shared-memory architecture using MEMSCALE

HTX connector to connect to the motherboard. This connector allows the processors in a motherboard to directly interact with our card as if it was a regular memory controller. Additionally, these cards, installed at every node, are also equipped with a network interface in order to be interconnected by using a 3D mesh topology.

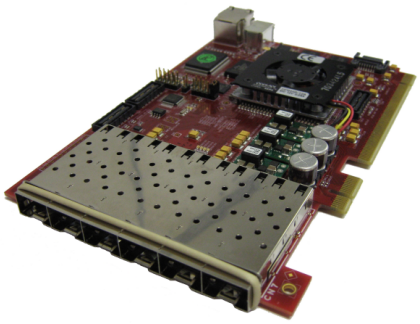

Figure 2: HTX card with an FPGA that implements the RMC functionality

By leveraging our proposal, processors can send a common read request (as a result of a *load* assembly instruction) to the RMC as if it was aimed to a regular memory module. However, in this case, the RMC has no memory modules directly attached to it but it must forward the read request to the appropriate remote node in the cluster. When the read request arrives to the remote node, the RMC in that node forwards the request to the corresponding local memory module, which replies to the RMC and then the data is sent all the way back to the processor that originated the memory operation. This is a simple mechanism that allows remote access with very low latency. Additionally, in order to scale this shared-memory architecture to virtually any system size, the new hardware does not provide coherency across nodes in the cluster. On the opposite, coherency is kept, when required, by the system software.

MEMSCALE allows database servers to scale out, that is, their capacity and performance can be progressively increased by adding new nodes to the cluster. Notice that scaling out regular database servers is usually accomplished by replicating an original server in order to distribute queries over a set of identical slaves and, therefore, allowing better throughput and lower latency as load per server is reduced. However, MEMSCALE follows a different approach, as adding

new nodes to a MEMSCALE cluster does not mean replicating the database, but providing more cores to access the shared-memory pool, thus increasing concurrency and therefore reducing average query latency. Actually, this mechanism could be seen as a way of scaling up the database server.

2.1 Database Porting

As our cluster architecture is presented to applications as a true shared-memory environment, they can be natively executed in it. Therefore, any database server with a main-memory storage engine can benefit from MEMSCALE. We chose MySQL as it is the most used open-source database server in the world and also because it offers a main-memory storage engine (known as *heap* or *memory*) that is very useful for us: we only have to extend this storage engine to use the global memory pool instead of using local main memory. To do so, the only thing we had to do was to slightly modify the dynamic memory allocation (instead of using a common *malloc* we used a library function with identical interface that allocates memory in the global memory pool). After that, as structures containing data and other synchronization information are on the memory pool, they are accessible by any processor in any node. As can be seen, porting a shared-memory application to MEMSCALE is straightforward.

3. RELATED WORK

In order to improve database server performance, it is becoming increasingly popular the use of Solid State Drives (SSD) to substitute the slow HDD as they offer more than one order of magnitude better results, as there are no mechanical parts to be moved. However, there is still the need for a software handler and, therefore, the related penalty is undergone. We can find two subclasses of this technology, SSD disks just used as an HDD and SSD memories directly attached to PCIe. Just to briefly compare against our proposal, according to our experiments with Kingston SSDs and a FusionIO PCIe card, SSD disks are one order of magnitude slower than MEMSCALE, while PCIe-based SSDs are only slightly slower than MEMSCALE (note that MEMSCALE is implemented in an FPGA prototype and its performance is far from the maximum one). Moreover, it is not only about latency but also about throughput: as MEMSCALE is able to execute queries in an entire cluster, to achieve a similar productivity a cluster should be equipped with a PCIe SSD card at each node, increasing its acquisition cost to a prohibitive level.

Other solutions intended to aggregate computing resources in a cluster are MySQL Cluster[2] and ScaleMP[4]. MySQL Cluster allocates data in main memory, as we do with MEMSCALE, but the way it is accessed (through a software socket) prevents good performance. According to our tests, a 16-node cluster with MySQL Cluster achieves a throughput two orders of magnitude lower than MEMSCALE. On the other hand, ScaleMP is a software that aggregates multiple nodes in a cluster to conform a single system image virtual machine. However, as it is software based, it presents a remote memory latency of 25us, while MEMSCALE presents a remote memory latency of 1.8us. Moreover, as we have experienced, ScaleMP quickly saturates in the presence of multiple threads reading the in-memory databases due to software bottlenecks, preventing this solution from achieving as good throughput as MEMSCALE.

4. DEMO DESCRIPTION

We have implemented our system proposal in a prototype cluster to show the feasibility of our idea and also to show some preliminary performance metrics. As it is a prototype implementation (mainly due to the fact that the RMC functionality in the HTX card is based on slow FPGA technology) we should be aware that the results will be below the potential ones if ASIC technology was used. However, as we will see in the demo, despite of the prototype status of MEMSCALE, it is currently achieving quite good results compared to other related solutions.

The central idea of the demo is to show the cluster at work by running a MySQL server on it. We have designed a simple database, depicted in Figure 3, that mimics the structure of a social network. The overall size of the database, including indexes, is 100GB.

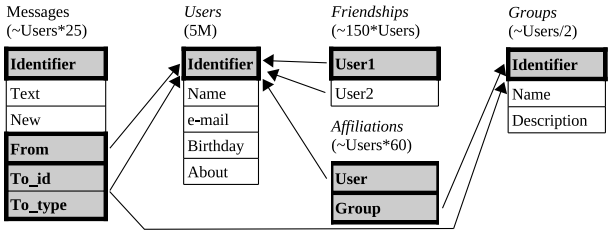

Figure 3: Table structure (gray background indicates an indexed field)

The demo allows to access the cluster in a remote way through a web interface, shown in Figure 4. Basically, we can launch a given quantity of queries against our prototype cluster. Although it is possible to execute any kind of SQL query, we have predefined six illustrative queries very typical of social networks that will emphasize the advantages of our system. In this way, we focus on sub-second queries even with this database size. Notice that these queries will walk through the tables exclusively by using indexed fields. Moreover, although the queries are rather simple, they present a by-nature erratic access pattern. This prevents the exploitation of data locality, especially in cluster contexts. For example, the query *"Get the names of the friends for a given user"* will perform a number of random accesses to the *users* table as the data on that table cannot be grouped in a way that satisfies locality for any given user. Therefore, when that table is distributed across different nodes in the cluster, the query will require data from all of them independently of the effort towards grouping friends together. On the live demonstration we execute thousands of this set of queries to get average query latency and cluster throughput in terms of queries per second.

Regarding the graphical appearance of the demo, during the execution of a given experiment we will receive feedback in real-time from our prototype cluster. The results of the different tests will be displayed in a 2D graph for comparing purposes. Additionally, the history of queries is displayed on the bottom left in order to easily compare them on the

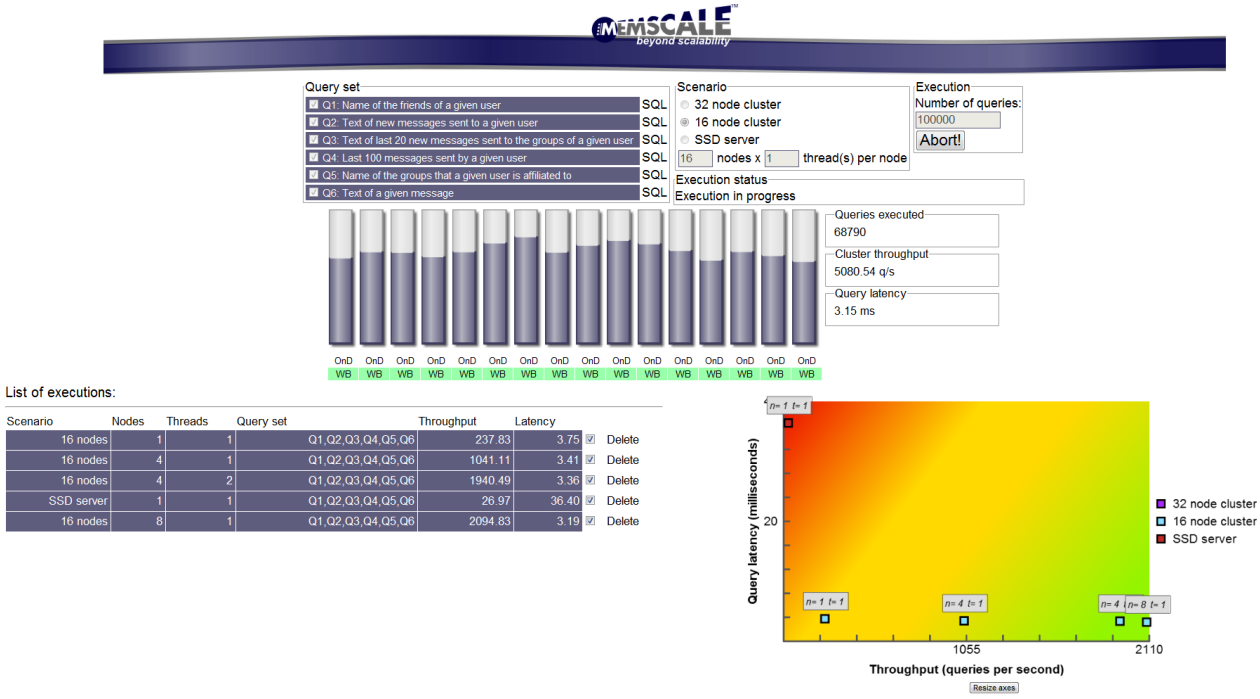

Figure 4: Screen shot of the demo's web page

2D graph on the right. In the central part of the window, the progress is displayed for all the nodes taking part in the execution of the current query.

The demo allows the user to execute queries in three different scenarios: a 16-node cluster, a 32-node cluster, and an SSD server. The first two scenarios will show our technology at work and will serve to show its scalability. We will see how query latency varies when the cluster size varies (together with throughput) because the number of nodes increases, and given a fixed link radix per node, the average distance in the network increases too and, consequently, query latency will also be affected. On the other hand, throughput linearly increases with cluster size, as more processors are available for executing queries (and more memory controllers are available for increasing global memory bandwidth). Our third scenario is an SSD-based server (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) that will serve to compare latency and throughput metrics. For the case of the SSD server, we will obviously not use the main-memory storage engine, but the MyISAM storage engine that allows good performance at reading.

For all the scenarios, the user can increase the number of query flows in terms of cores executing queries. It is expected linear improvement, in queries per second, as the number of queries served in parallel increases. However, during the demo we will see how the SSD-based server saturates well before every core in the motherboard is being used. On the contrary, MEMSCALE will show better scalability as the design of a motherboard ensures perfect matching between cores and main memory capabilities. Moreover, this kind of test will show how much inter-node traffic affects global throughput and latency.

Additionally we are able to reduce processor frequency and see how cluster performance is almost not impacted. According to our studies, we will see how processor frequency is not critical in terms of query latency or database throughput. However, a reduction in CPU frequency (2GHz to 1GHz) can save up to 18% of energy and up to 20% of power across the entire cluster.

Finally, in the following URL: *http://www.memscale.eu/CIKM_demo* it can be found a video that contains a demo preview showing the different steps on the demonstration and some of the multiple tests that can be carried out.

5. REFERENCES

[1] MEMSCALE: New Possibilities for Shared Memory in Clusters . `http://www.hpcadvisorycouncil.com/events/2011/european_workshop/pdf/14_U_Valencia.pdf`.

[2] MySQL Cluster. `http://www.mysql.com/products/cluster`.

[3] Oracle TimesTen In-Memory Database. `http://www.oracle.com/us/products/database/timesten`.

[4] ScaleMP. `http://www.scalemp.com`.

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

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

[7] B. Fitzpatrick. Linux Journal: Distributed Caching with Memcached. 124:72–76, 2004.