Software-Distributed Shared Memory over Heterogeneous Micro-Server Architecture

Loïc Cudennec

Software-Distributed Shared Memory over Heterogeneous Micro-Server Architecture

Loïc Cudennec

CEA, LIST, Saclay, France
loic.cudennec@cea.fr

Abstract. Nowadays, the design of computing architectures not only targets computing performances but also the energy power savings. Low-power computing units, such as ARM and FPGA-based nodes, are now being integrated together with high-end processors and GPGPU accelerators into computing clusters. One example is the micro-server architecture that consists of a backbone onto which it is possible to plug computing nodes. These nodes can host high-end and low-end CPUs, GPUs, FPGAs and multi-purpose accelerators such as manycores, building up a real heterogeneous platform. In this context, there is no hardware to federate memories, and the programmability of such architectures suddenly relies on the developer experience to manage data location and task communications. The purpose of this paper is to evaluate the possibility of bringing back the convenient shared-memory programming model by deploying a software-distributed shared memory among heterogeneous computing nodes. We describe how we have built such a system over a message-passing runtime. Experimentations have been conducted using a parallel image processing application over an homogeneous cluster and an heterogeneous micro-server.

Keywords: S-DSM, Data Coherence, Heterogeneous Computing

1 Introduction

Heterogeneity is slowly entering high-performance computing. After a decade figuring out how to cope with mixed CPU and GPU nodes for performance at both the hardware and software levels, new requirements now concern the limitation of the power consumption. Low-power CPUs (ARM) and accelerators (manycore, FPGAs) are joining the computing resource list. These resources can run regular tasks in a massively parallel way, while keeping the electricity bill reasonable. Micro-servers have been developed in this direction. They offer a communication and power supply backbone onto which it is possible to plug heterogeneous computing and data storage nodes. These nodes can host regular CPU such as Intel i7, clusters of ARM Cortex (more popular in smartphones than in HPC) and Xilinx/Altera FPGAs to deploy specific IPs. But the micro-server architecture comes with a price: it escalates the problem of managing the heterogeneity of resources. Current approaches include hybrid programming such

as MPI/OpenMP (message passing between nodes and parallel programming within nodes) and task-based models such as OpenCL, StarPu and OmpSs that encapsulate the user code into a specific framework (kernels, tasks, dataflow). These systems have been ported to different processor architectures, even on FPGAs for OpenCL, addressing the heterogeneity of the platforms. Unified distributed memory systems can be built on top of heterogeneous platforms using, for example, cluster implementations of OpenMP and PGAS implementations (provided it does not rely on hardware mechanisms such as RDMA). In this work, we explore the possibility of deploying a full software-distributed shared memory system to allow MPMD programming on micro-servers (a distributed architecture with heterogeneous nodes). This is quite new for such systems, for two reasons: First, there is a lack of specification and formalization against hardware shared memory, and also because of a potential scaling problem. Second, software shared memory, while being famous with computing grid and peer-to-peer systems, is seen as a performance killer at the processor scale. We think that micro-servers are standing somewhere in-between: from the multi-processors they inherit the fast-communication links and from the computing grids, they inherit the heterogeneity, the dynamicity of resources and a bit of scaling issues. In this work, we propose an hybrid approach where data coherency is managed by software between nodes and by regular hardware within the nodes. We have designed and implemented a full software-distributed shared memory (S-DSM) on top of a message passing runtime. This S-DSM has been deployed over the RECS3 heterogeneous micro-server, running a parallel image processing application. Results show the intricacies between the design of the user application, the data coherence protocol and the S-DSM topology and mapping. The paper is organized as follows: Section 2 describes some micro-server architectures and the way they are used. Section 3 presents the S-DSM. Section 4 describes the experiments on both homogeneous and heterogeneous architectures. Section 5 gives some references on previous works. Finally, section 6 concludes this paper and brings new perspectives.

2 Micro-servers and Heterogeneous Computing

Micro-servers such as HP Moonshot [1] and Christmann RECS [10,7] are modular architectures that can be adapted to a particular application domain. As illustrated by Figure 1, a chassis provides power supply, cooling systems, as well as a backplane that hosts several integrated networks (management, computing..) and a set of slots to plug computing boards (also called servers). These computing boards share the same interface and form factor (for example COM Express). However, the inner design is quite free, which is source of heterogeneity with important unbalance in computing performance and communication speed. Such architecture is known to reduce power consumption, save space and avoid cable spaghetti. Data management depends on the configuration of the micro-server: we assume that there is at least one CPU per node that is able to run a full operating system and locally store data, either on physical memory

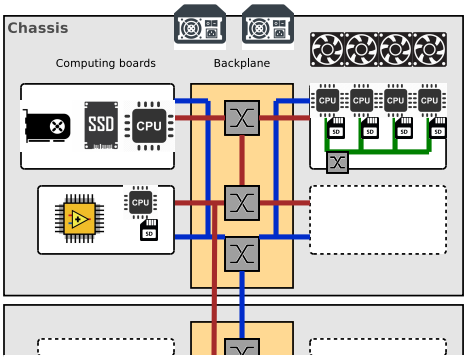

Fig. 1: Micro-server hosting nodes with CPU, low-power CPU, GPU and FPGA.

or on disk (SSD, SD card). On this type of distributed architecture, data are usually managed using message passing or remote accesses that do not take into account the heterogeneity of the storage medium. Furthermore, the user is in charge of the localization and the transfer of data. We think that there is room for some improvements in data management over such platforms. S-DSM can be used to transparently federate memories of the computing boards and offer an abstraction of the storage at the global scale. However, as far as we know, S-DSM are mainly designed for homogeneous platforms (except for the communications when deploying over NUMA architectures), and they have not been deployed over micro-servers. In this work we deploy an in-house S-DSM over a micro-server. We analyze what are the limitations of the approach and propose some improvements for future S-DSM deployments.

3 Software-Distributed Shared Memory

The Software-Distributed Shared Memory (S-DSM) interfaces user applications relying on the shared memory programming model to a given hardware architecture in which physical memories can be distributed. With this system, the application is written as a set of threads/tasks from which it is possible to allocate and access shared data (close to the *Posix* and *shmem* models). To perform such accesses we have defined an API inspired by the entry consistency model [5]. Portion of codes that access a shared data are protected between *acquire* and *release* instructions applied to the data. There are two *acquire* instructions to discriminate a shared access against an exclusive access (multiple readers, single writer). The API also provides *rendez-vous* and other synchronization primitives. The logical organization of the S-DSM follows a client-server model. A client runs the user code, as well as some S-DSM code (mainly hidden behind the *malloc*, *acquire* and *release* instructions). The server only runs S-DSM code and is used to manage metadata and store data. Each client is attached to at least one server. The resulting topology can be compared to the super-peer topology

found in large distributed systems. *Chunks* are the atomic piece of data managed by the S-DSM. The size of the chunk can be set by the application. Whenever a data is allocated in the shared memory, if the size is larger than the chunk size, then it will allocate more than one chunk. The memory space allocated on the client is always a contiguous space on which it is possible to use pointer arithmetic. However, on the server side, the chunks are managed independently and can be spread among the servers in any order. *Chunks* can be compared to pages in operating systems and so-called *chunks* in peer-to-peer systems. Each *chunk* is under the control of a data coherence protocol. The S-DSM allows several coherence protocols to run concurrently, but not for the same chunks. The coherence protocol is in charge of the localization and the transfer of the chunk. Each protocol implements the actions to execute whenever *acquire* and *release* instructions are called on the client side, and it also implements a distributed automata for the servers. The home-based MESI protocol [8] is an an example of a widely-used cache coherence protocol for multi-core processors. *Home-based* means that the management of each chunk, including metadata, is the responsibility of one server called *home-node*. The home-node does not necessarily store the data. Home-nodes are usually assigned to chunks using a round-robin arrangement. MESI is one of the protocols that has been implemented in the S-DSM. In this paper we only refer to this protocol. We have implemented an ANSI C version of such a S-DSM using the OpenMPI message passing runtime. There is a weak dependence on OpenMPI as it only uses *send* and *receive* primitives (no collective functions for example), and it is quite straightforward to switch to another MP middleware. However, the MPI runtime is convenient because it handles the deployment and the bootstrap of tasks and can be installed in many Linux distributions, which is a serious argument when deploying on an heterogeneous platform. The implementation of the S-DSM is roughly $12k$ lines of code, including data coherence protocols.

4 Experiments with an Image Processing Application

The S-DSM has been deployed over two testbeds: an homogeneous cluster of desktop computers and a heterogeneous micro-server. Descriptions of testbeds are given in Figure 2. The purpose of these experiments is to highlight the behavior of the S-DSM runtime and the home-based MESI coherence protocol. This is why some choices regarding the S-DSM setup such as the granularity of the data and the topology are more set to stress the system rather than to get performance. All experiments use the exact same S-DSM and application codes, and the same input data. Only the description of the topology and the placement of tasks (MPI *rankfile*) differ.

4.1 Parallel Eager-Scheduled Convolution Application

The convolution application is an image processing application that calculates for each pixel of the input image a new value based on the surrounding pixels

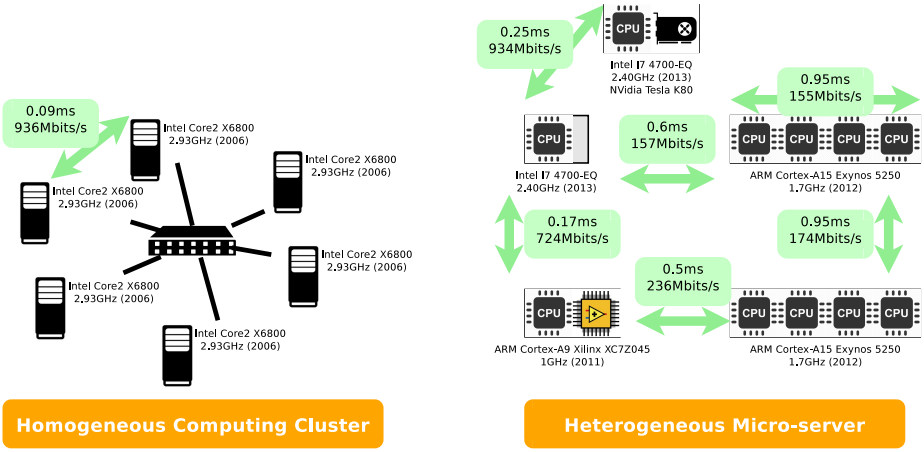

Fig. 2: Testbeds used for the experimentations: an homogeneous cluster of desktop computers and an heterogeneous Christmann RECS 3 Antares Box Microserver. Latencies are given by *Ping* and throughputs by *Iperf*. If not specified, we assume roughly the same performances as similar links.

(stencil) multiplied by some coefficients (kernel). For example, some stencil and kernel combinations can be used for edge detection. A parallel version of the code is straightforward and, because each pixel can be processed independently (the result does not depend on other results), there is no constraint on granularity: pixels, lines or macro blocks can be processed concurrently. We have implemented this algorithm using an eager scheduling strategy on top of the S-DSM. The eager strategy works as follows: a set of jobs is shared between tasks. Each task concurrently iterates on the next available job. Tasks that are running faster will process more jobs. This is an interesting property for running a parallel application onto heterogeneous resources: if the jobs are equally splitted between tasks then the tasks that are running on the most powerful resources will have to wait for the weakest one. Instead, eager scheduling allows load balancing and makes resources busy at -almost- all time. We have set the granularity of the parallel computation to the image line size and we use the same size for the S-DSM chunk size. Therefore, the input and output images (as well as the intermediate representation - this is a 2-step algorithm with a convolution followed by a normalization) are represented by a set of chunks, one chunk per line. A job consists of processing one line. The concurrency comes from the convolution kernel size that requires to read three contiguous lines to process the central line and a possible overlapping with other tasks. Shared data also include the available jobs vector and the current max pixel value found while applying the convolution, and used for normalization. All shared data are accessed under the control of the home-based MESI protocol. Experimentations are based on the same code, using a 3.7MB 2560x1440 grayscale image as input. This image size is large enough to get tangible results on the behavior of the application, and the

granularity is small enough to stress the S-DSM and see what are the bottlenecks (in fact the granularity is far too small to get any speedup and most of the time is spent into S-DSM mechanisms and communications). In the experiments, the amount of messages received by the main memory server, including the S-DSM bootstrap and the consistency protocol goes from 30000 to 112000 messages in a single run, which explains the poor performances. The application is composed of 3 main roles: at least one memory server, one and only one i/o task that copies the image from disk into the memory, waits for the end of calculation, and copies back the result from memory to disk, and at least one processing task. This makes possible to deploy different topologies of the same application. The minimal topology being one memory, one i/o and one processing task. This latter topology is used to bench the different CPUs of the testbeds with the following results: 1.4s for i7-5600U, 2s for i7-4700EQ, 3.2s for Core2-X6800, 7.6s for Cortex-A15 and 35.7s for Cortex-A9. All processing times are *real* values given by the Unix *time* command and therefore include the OpenMPI runtime bootstrap, the S-DSM bootstrap and the disk accesses to the input and output files. The important gap between the Intel i7-5600U and the ARM Cortex-A9 is also explained by the disk technology: a SSD for the i7 and a SD card for the Cortex-A9. In that context, deploying a S-DSM over heterogeneous nodes can be used to pin i/o tasks onto nodes with high-speed disks and keep all data in memory otherwise.

4.2 Homogeneous Cluster Architecture

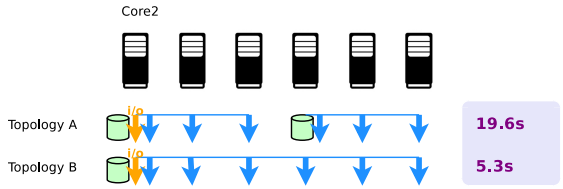

Fig. 3: Processing time on the cluster using different S-DSM topologies. The light-green cylinders represent memory servers and the arrows represent the clients. Orange clients are input/output tasks while blue clients are processing tasks. The horizontal blue lines define the memory clusters (to what server is connected each client).

Before deploying the S-DSM onto the RECS3 micro-server, we use a homogeneous computing cluster with different application topologies. The goal is to observe the performance variations and determine if it comes from the S-DSM implementation or from the heterogeneity of the resources. Figure 3 shows the processing times for two topologies running on 6 nodes. Topology A is made of two memory clusters, three processing tasks in each memory cluster and one

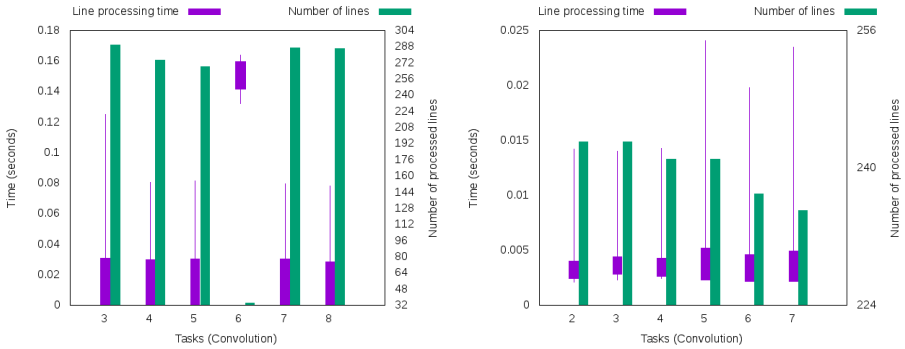

Fig. 4: **Left: Topology A**. Memory servers are collocated with tasks 3 and 6. The i/o task is collocated with task 3. **Right: Topology B**. The memory server and the i/o task are collocated with task 2.

i/o in one memory cluster. Topology B is a single memory cluster hosting six processing tasks and the i/o task. A runs almost 4 times slower than B: adding a memory server brings complexity in the data management: more control and data messages, as well as one additional MPI process that does not contribute to the job. The benefit of adding a new cache does not hide this overhead. Left Figure 4 gives the minimum, maximum and standard deviation of the line processing time, as well as the number of processed lines for each computing task of topology A. Task 6 is performing badly, because of the activity of the collocated memory server. Despite a collocated memory server, task 3 has no performance drop because it directly benefits from the local cache that has been filled by the collocated i/o task. Right Figure 4 presents the same metrics for topology B in which it appears that performances are now inline with the homogeneous cluster architecture. One conclusion at this step of experimentation is that the application topology must be tightly chosen according to the application behavior and the underlying hardware. In this particular scenario, adding a zealous cache is not an option.

4.3 Heterogeneous Micro-server Architecture

In this set of experiments we deploy the application onto the RECS3 micro-server as presented in Figure 2, except that we use only one i7 node out of the two. We deploy four different topologies, as presented in Figure 5. In topology C, Cortex-A9 (the weakest node regarding computing power) is discarded. We take the results as reference to study the influence of this particular node in the other topologies. Top-left of Figure 6 shows the performance of each computing task. Despite the heterogeneity of the hardware, all tasks achieve quite similar performances. The MESI data coherence protocol implementation is designed for homogeneous architectures, in which distributed roles share the same duty. Metadata management is spread across the i7 and Cortex nodes and one access

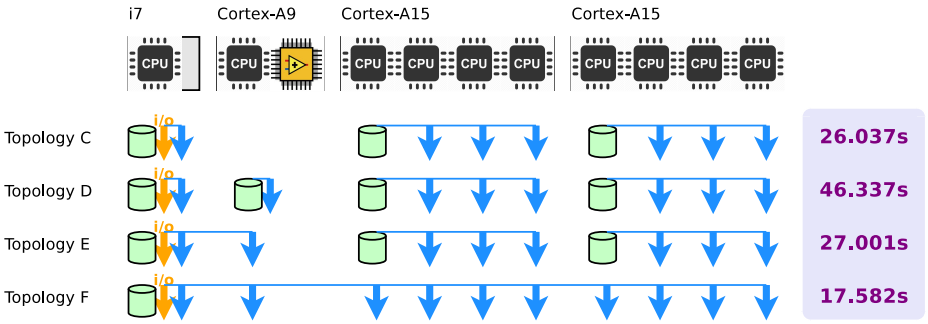

Fig. 5: Processing time on the RECS3 micro-server using different S-DSM topologies.

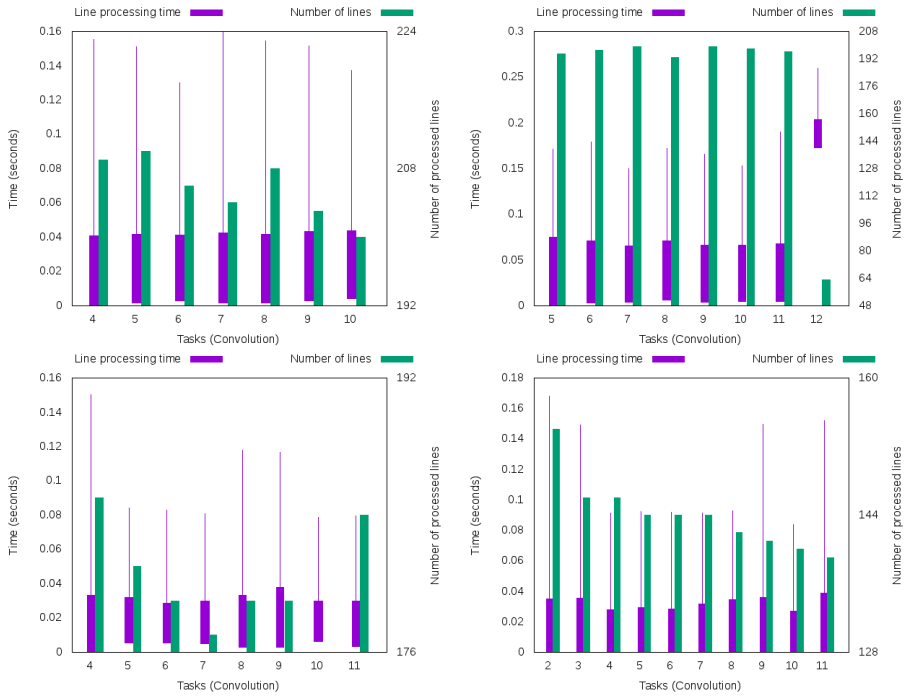

Fig. 6: **Top-left: Topology C**. Task 4 is running on the i7 processor, tasks 5 to 10 on Cortex-A15 processors. **Top-right: Topology D**. Task 5 is running on the i7 processor, tasks 6 to 11 on Cortex-A15 processors and task 12 on Cortex-A9. **Bottom-left: Topology E**. Task 4 is running on the i7 processor, tasks 5 to 10 on Cortex-A15 processors and task 11 on Cortex-A9. **Bottom-right: Topology F**. Task 2 is running on the i7 processor, tasks 6 to 10 on Cortex-A15 processors and task 11 on Cortex-A9.

to shared data on the i7 can trigger some requests to a Cortex node in charge of the data, and vice-versa. In this experiment the processing time of a line is mainly spent in getting access to the data. And this time has to be paid by all tasks, whatever the resource they are running on. Topology D collocates a memory server and a processing task on the Cortex-A9. While it adds a new worker to the application, it also adds a new server that will be responsible of managing some metadata. This is probably too much to handle for such CPU, as shown by top-right Figure 6: the overall computing time is almost twice the time than without Cortex-A9 and task 12 runs slower than the other.

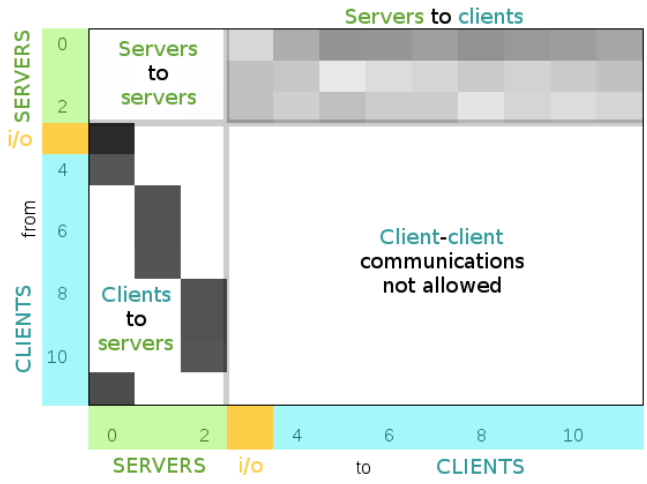

Fig. 7: **Topology E**. Communication heatmap. Each cell represents the cumulative size of messages that have been sent *from* tasks indexed vertically, *to* tasks indexed horizontally. Tasks 0 to 2 are memory servers. Task 3 is the i/o client. Tasks 4 to 11 are processing tasks. Values are normalized to grayscale, darker is bigger.

With topology E, the memory server is removed from Cortex-A9 and the remaining processing task (still running on Cortex-A9) is attached to the memory server located on the i7 node. This is the best scenario for Cortex-A9 because 1) it interacts directly with the memory server running on the most powerful resource and 2) the network connectivity is far better than with the Cortex-A15 nodes (0.17ms, 724Mbits/s versus 0.5ms, 236Mbits/s). The overall computing time is quite comparable with the C scenario: running a memory server on the Cortex-A9 was a terrible choice. Bottom-left Figure 6 reveals that task 11 located on Cortex-A9 performs as good as tasks located on Cortex-A15 and has even be able to take more jobs than the other. The communication heatmap for topology E

is given in Figure 7. Communications between servers are quite light and mainly consist of a large number of very small control messages. This would be quite different in the case of a cache cooperative protocol. Communications from clients to servers strictly follow the topology description: a client only sends messages to the memory server it is attached to. The important traffic corresponds to messages for updating chunks on the server after completing jobs. Servers to clients communications consist of a mix of control and data messages. It shows that the memory server 0 located on the i7 node has sent more data to the clients than the two other servers located on the Cortex-A15 nodes. Finally, clients to clients communications are not allowed in this protocol (this optimization is not implemented at this time). Topology F is made of one memory server located on the i7 node and ten processing tasks (one per CPU). As for topology B this strategy gives better performance, but the improvement over topology C is not that important than with the homogeneous testbed. Bottom-right Figure 6 shows that all processing tasks are now performing at the same speed, hiding the resource computing power they are running on.

4.4 Discussions

The S-DSM runtime has a major influence on the performance map of the application: we have shown that running over an heterogeneous architecture can lead to a global overhead in which computing tasks deployed on the most powerful processors cannot perform better than tasks deployed on weaker processors. This is mainly due to the home-based MESI coherence protocol implementation that equally balances the metadata management on memory servers. In this context, there is a performance fall when clients access shared data that are managed by a server running on a weak resource. And this is the case with the convolution application in which the management of lines is spread among the memory servers and the number of shared accesses is the same for all lines. Therefore, data coherence protocols should be designed with the possibility to adapt the metadata management load depending on the resource performances (computing power and network). In this direction, we can propose the dissociation of the data management (metadata) and the cache system. For example, in this paper experiments, the whole metadata management could be handled by the most powerful node while several data-only caches could be spread among other nodes. Another aspect is the importance of the placement (and possibly the routing) of the data coherence protocol roles onto the resources. A key aspect is the collocation of roles (and user tasks) that need to extensively communicate. In most of the message passing runtime implementations, such communications are locally optimized. Placement should be planned using offline static analysis and/or using dynamic mechanisms. In this paper we have proposed arbitrary topologies. We think that a more automated approach should be used, possibly with the help of operational research algorithms. Finally, one of the main purpose of the micro-server architecture is to offer computing power with interesting performance per watt compared to regular computing clusters. Some topologies

might not be adapted to reach the best execution time, but could provide interesting properties regarding energy consumption. And in some scenarios the energy consumption might be a more valuable metric.

5 Related Works

Software-Distributed Shared Memory has become popular in the late eighties [11] with the introduction of systems for computing clusters [5,6,2,3], followed by systems for computing grids [4,12] and many-core processors [14]. These S-DSM are designed for a particular architecture and reasonably expect the same performance from the physical resources. Deploying S-DSM over heterogeneous systems has been studied in 1992 with Mermaid [15] and Jade [13] running on SPARC, DEC and DASH-based machines. With Mermaid, the authors focus on the problem of data conversion between processors. While both systems are undoubtedly a demonstration of a S-DSM running over an heterogeneous architecture, the conclusions only highlight the functional side of the approach. Later on, with the Asymmetric-DSM [9], the authors propose a data coherence protocol that is specific to asymmetric links between host CPU and accelerators. The work presented in this paper not only demonstrates the possibility of deploying a S-DSM over a state-of-the-art micro-server architecture. It also focuses on the intricacies between the S-DSM runtime, the data coherence protocol and the application behavior.

6 Conclusion

Low-power architectures are now entering high-performance computing systems. Micro-servers are one example of such integration, with a potentially high level of heterogeneity between computing nodes. Message passing and dataflow are natural programming paradigms that come into mind in order to exploit the architecture. We think that shared memory can also helps by providing a convenient abstraction layer between the application and the data storage systems. In this paper we have shown that a software-distributed shared memory can also be deployed on micro-servers. It also shows that the price to pay is a tight study of the S-DSM core functions, choosing or adapting the right data coherence protocol and profiling the application regarding shared data accesses.

Acknowledgments. This work received support from the H2020-ICT-2015 European Project M2DC - Modular Microserver Datacentre - under Grant Agreement number 688201.

References

1. Hpe serveur moonshot. Hewlet Packard Entreprise, https://www.hpe.com/us/en/servers/moonshot.html

2. Amza, C., Cox, A.L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., Zwaenepoel, W.: TreadMarks: Shared memory computing on networks of workstations. IEEE Computer 29(2), 18–28 (Feb 1996)
3. Antoniu, G., Bougé, L.: Dsm-pm2: A portable implementation platform for multithreaded dsm consistency protocols. In: Proceedings of the 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments. pp. 55–70. HIPS '01, Springer-Verlag, London, UK, UK (2001)
4. Antoniu, G., Bougé, L., Jan, M.: JuxMem: an adaptive supportive platform for data-sharing on the grid. Scalable Computing: Practice and Experience (SCPE) 6(3), 45–55 (Nov 2005)
5. Bershad, B.N., Zekauskas, M.J., Sawdon, W.A.: The Midway distributed shared memory system. In: Proceedings of the 38th IEEE International Computer Conference (COMPCON Spring '93). pp. 528–537. Los Alamitos, CA (Feb 1993)
6. Carter, J.B., Bennett, J.K., Zwaenepoel, W.: Implementation and performance of Munin. In: 13th ACM Symposium on Operating Systems Principles (SOSP). pp. 152–164. Pacific Grove, CA (Oct 1991)
7. Cecowski, M., Agosta, G., Oleksiak, A., Kierzynka, M., v. d. Berge, M., Christmann, W., Krupop, S., Porrmann, M., Hagemeyer, J., Griessl, R., Peykanu, M., Tigges, L., Rosinger, S., Schlitt, D., Pieper, C., Brandolese, C., Fornaciari, W., Pelosi, G., Plestenjak, R., Cinkelj, J., Cudennec, L., Goubier, T., Philippe, J.M., Janssen, U., Adeniyi-Jones, C.: The m2dc project: Modular microserver datacentre. In: 2016 Euromicro Conference on Digital System Design (DSD). pp. 68–74 (Aug 2016)
8. Culler, D., Singh, J., Gupta, A.: Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann, 1st edn. (1998), the Morgan Kaufmann Series in Computer Architecture and Design
9. Gelado, I., Stone, J.E., Cabezas, J., Patel, S., Navarro, N., Hwu, W.m.W.: An asymmetric distributed shared memory model for heterogeneous parallel systems. In: Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems. pp. 347–358. ASPLOS XV, ACM, New York, NY, USA (2010)
10. Griessl, R., Peykanu, M., Hagemeyer, J., Porrmann, M., Krupop, S., v. d. Berge, M., Kiesel, T., Christmann, W.: A scalable server architecture for next-generation heterogeneous compute clusters. In: 2014 12th IEEE International Conference on Embedded and Ubiquitous Computing. pp. 146–153 (Aug 2014)
11. Li, K.: IVY: a shared virtual memory system for parallel computing. In: Proc. 1988 Intl. Conf. on Parallel Processing. pp. 94–101. University Park, PA, USA (Aug 1988)
12. Nicolae, B., Antoniu, G., Bougé, L., Moise, D., Carpen-Amarie, A.: Blobseer: Next-generation data management for large scale infrastructures. J. Parallel Distrib. Comput. 71, 169184 (February 2011)
13. Rinard, M.C., Scales, D.J., Lam, M.S.: Heterogeneous parallel programming in jade. In: Proceedings Supercomputing '92. pp. 245–256 (Nov 1992)
14. Ross, J.A., Richie, D.A.: Implementing openshmem for the adapteva epiphany risc array processor. Procedia Computer Science 80, 2353 – 2356 (2016), international Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA
15. Zhou, S., Stumm, M., Li, K., Wortman, D.: Heterogeneous distributed shared memory. IEEE Transactions on Parallel and Distributed Systems 3(5), 540–554 (Sep 1992)