Managing Cache Memory Resources in Adaptive Many-Core Systems

Gustavo Girão[1(✉)] and Flávio Rech Wagner[2]

[1] Digital Metropolis Institute, Federal University of Rio Grande do Norte, Natal, Brazil
girao@imd.ufrn.br

[2] Informatics Institute, Federal University of Rio Grande do Sul, Porto Alegre, Brazil
flavio@inf.ufrgs.br

Abstract. In the last decades, the increasing amount of resources in embedded systems has been leading them to the point where an efficient management of these resources is mandatory, especially for the memory subsystem. Current MPSoCs have more than one application running concurrently. Hence, it is important to identify the memory needs of these applications and provide them accordingly. In this work we propose the use of a cluster-based, resource-aware approach to provide this efficient environment. The solution proposed here improves the overall performance of these systems by aggregating memory resources in clusters and redistributing these resources among applications based on a fairness criterion. For this memory clustering proposal, we use the information of external memory access-es as an estimate of the amount of memory required by each application. T Experimental results show that, depending on how the redistribution of memory resources among application occurs, the overall system can improve performance up to 18% and the energy savings can reach up to 20%.

Keywords: Many-core · Resource management
Adaptable memory hierarchy · Network-on-chip

1 Introduction

The memory resources in many-core systems must be even more abundant than processing elements. As a classical and perennial bottleneck of computer systems, the memory resources must be managed with the outmost efficiency in order to provide decent performance.

The aggregation of resources as well as their efficient distribution among concurrent applications may lead to substantial gains for the overall system. It is a well-known fact that applications may have different needs of processing and memory requirements. Processing requirements refer to the needs of the application to execute its code faster, for instance using TLP (Thread Level Parallelism) and ILP (Instruction Level Parallelism) techniques. As a usual solution,

© IFIP International Federation for Information Processing 2017
Published by Springer International Publishing AG 2017. All Rights Reserved
M. Götz et al. (Eds.): IESS 2015, IFIP AICT 523, pp. 172–182, 2017.
https://doi.org/10.1007/978-3-319-90023-0_14

processing demands must be addressed by assigning more processor elements (e.g. cores or specialized hardware) and/or more time slices to applications. On the other hand, certain applications manipulate considerably higher amounts of data and, therefore, the use of more memory resources (e.g. L1/L2 caches) would increase their efficiency. Figure 1 shows the memory demands of applications from the SPEC benchmark [1]. For these applications, it is possible to see that the amount of memory access instructions (read or write) can range from 30% to 70% of all instructions. It is clear that an efficient memory resource distribution system, in an environment that can run several applications like those, can be advantageous. In this work, we explore various aspects of a memory resource management approach in an NoC-based MPSoC platform used as case study. First, we introduce a mechanism that is able to assign each L2 memory bank in the system to the exclusive use of a given application (and its respective tasks). This mechanism is based on dynamically changeable association tables present in every L1 cache, which indicate the L2 cache banks assigned to the application, their memory address range, and their NoC addresses. Second, we establish a mechanism to redistribute the L2 cache banks among the applications. Experimental results show that an overall system improvement can be reached at the expense of the performance of some individual applications. The key observation is that some applications have more impact on the overall system execution than others, and, consequently, reducing the memory latency of these critical applications can lead to performance gains. Results show that the overall execution time decreases up to 18% and energy savings reach up to 20% in the best case.

The remaining of this paper is organized as follows. The next section presents the baseline architecture considered in this work and the Memory Clustering mechanism itself, including the redistribution approach and redistribution mapping. Section 3 exposes the experimental setup, while Sect. 4 presents the results. Related work is discussed in Sect. 5, and, finally, in Sect. 6 conclusions are drawn and future work is outlined.

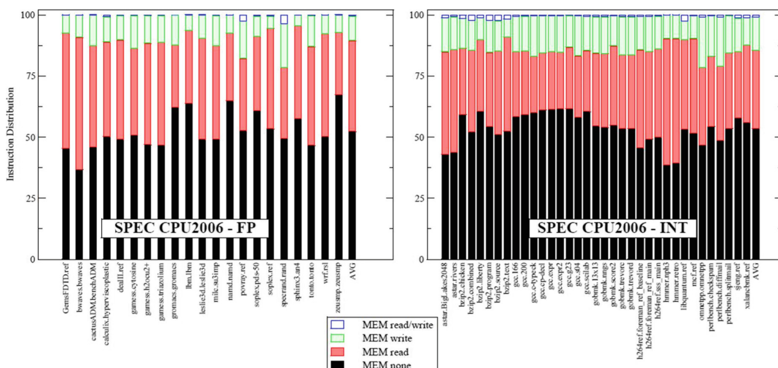


Fig. 1. Memory requirements from SPEC. [1]

2 Memory Clustering

Let us consider the baseline architecture for this mechanism as an NoC-based MPSoC whose nodes are as presented in Fig. 2. In this case, the memory subsystem is homogeneous and composed of local private L1 caches and distributed shared L2 caches. Inside each of the four nodes in the corners, there is also an external memory controller, which has a very important role in this approach. The key idea of this methodology, called Memory Clustering, is to assign the L2 memory blocks according to the needs of each application. The intuition here is that reserving more memory resources to applications that deal with more data will lead to a faster execution overall.

It is important to notice that, as the goal is to improve the *overall* performance and energy savings, the Memory Clustering approach must act on the address spaces of all applications (through the L2 caches) instead of simply redistributing the L1 cache resources, which apply only to the address space of the application currently running in the processor to which each L1 is attached.

The mechanism proposed in this work is a cluster-based resource-aware approach for the memory subsystem. In a previous work [2], this cluster-based approach for resource management has been introduced. However, only processing resources have been considered. In this paper we consider a cluster as an aggregation of physical resources (more specifically, L2 caches) delegated to an application. More than one application may be running in the system, and each one has its own set of resources. Due to the importance of the memory subsystem to the overall system performance, it is desirable to reduce memory latency for data-intensive applications.

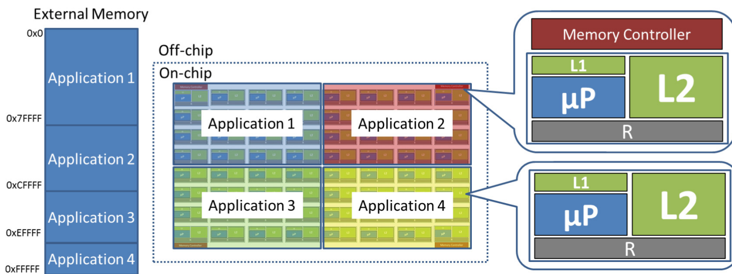


Fig. 2. Example of data space for a set of applications.

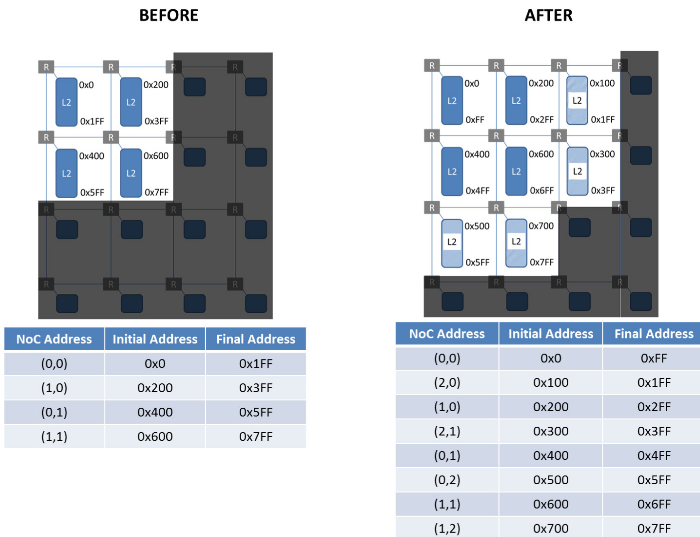
The proposed process for redistributing memory resources occurs as follows. Since the beginning of the execution, the memory controllers count the number of external memory accesses made by each application cluster (each application cluster being the aggregation of resources initially assigned to each application, as shown in Fig. 2). At some point during the system execution, these memory controllers (located in the corners of the MPSoC, as presented in Fig. 2) use these statistics to define which application has more cache misses. A synchronization step is performed when one of these memory controllers (henceforth known as master controller) receives messages from the other controllers informing the number

of cache misses of the various applications. After that, this master controller uses some redistribution policy (to be detailed in the next sections) to establish how many memory resources (L2 memory banks) each application cluster should have. This number of L2 memory banks allocated to each cluster will be proportional to the percentage of cache misses from the corresponding application. This strategy is an attempt to reproduce, at runtime, the knowledge presented in Fig. 4, which shows, for a system concurrently running four applications, the amount of memory accesses extracted from an application profile at design time.

In order to redistribute memory resources, this approach takes advantage of a previously introduced directory-based cache coherence mechanism [3]. This mechanism uses a small table called ATA table on each cache in the system, which relates an address range to the memory module in the system that may have blocks that belong to such range. This means that each cache memory only perceives a certain amount of physical memory modules in the system, depending on the memory address range of its current memory blocks.

As explained before, at the beginning of the execution each application cluster has its own set of L2 caches and all L1 caches in this cluster have an ATA table that indicates which range of addresses is possibly available in each L2. Each L2 cache controller also knows which addresses should be placed in its own memory.

Let us assume a small scenario of only four L2 caches for an application. As presented in Fig. 3, each cache bank can have a certain contiguous range of addresses from the external memory address space. On the left side of the figure, there is a representation of the ATA table of L1 caches in the cluster assigned to this particular application, where each line corresponds to the range of addresses assigned to each node of the cluster.



NoC Address	Initial Address	Final Address
(0,0)	0x0	0x1FF
(1,0)	0x200	0x3FF
(0,1)	0x400	0x5FF
(1,1)	0x600	0x7FF

NoC Address	Initial Address	Final Address
(0,0)	0x0	0xFF
(2,0)	0x100	0x1FF
(1,0)	0x200	0x2FF
(2,1)	0x300	0x3FF
(0,1)	0x400	0x4FF
(0,2)	0x500	0x5FF
(1,1)	0x600	0x6FF
(1,2)	0x700	0x7FF

Fig. 3. ATA table and MPSoC before and after resource distribution.

After the synchronization step, as illustrated in Fig. 3, the master controller may reach to the conclusion that this cluster needs four additional memory banks. After choosing which memory banks in the system shall be aggregated to this cluster (the dashed L2 cache banks represent these recently added resources), the ATA tables must be updated. On the bottom-right side of Fig. 3, the new ATA table is illustrated. It is important to notice that the addresses from the application address space are equally divided among the L2 cache banks.

3 Redistribution Policies and Mapping

Once established how the number of L2 caches associated to each L1 cache can be modified, the next decision in the memory clustering mechanism regards the moment when resources (in this case, L2 caches) are taken/given from/to a cluster, thus creating a resizable cluster. Two main approaches are presented here: *Pre-defined Distribution* and *On-demand Distribution*.

The Pre-defined distribution is the model used as baseline for the experimental results. The idea is, based on a design time profiling of the application, to establish beforehand the number of L2 memory modules that each application should have. This distribution takes into account the amount of input data for each application. Since external memory accesses can be very costly, the goal here is to minimize this situation by giving more cache memory space for the most data-intensive applications. This is a static offline approach and is used only to establish how good the second approach is. The graph on Fig. 4 presents the amount of memory accesses for each application used in the experiments and for each scenario with different MPSoC sizes (number of nodes). The memory resources (in this case, L2 cache memory banks) can be divided among the applications proportionally to the amount of data that each one handles.

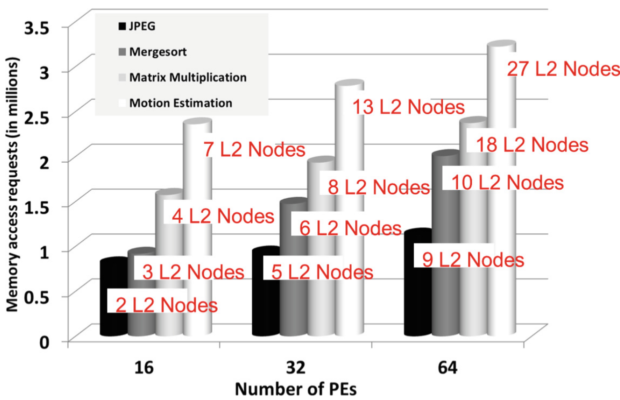


Fig. 4. Pre-defined distribution.

However, this information depends on profilling and it may not always be available. Therefore, some runtime approach should be considered. In the On-demand distribution, all clusters initially use the same amount of L2 caches.

After a certain amount of cycles, the external memory cache controllers (located on the corners of the MPSoC) exchange information, and, based on the number of external memory accesses, the master controller defines which clusters need more memory nodes. The master memory controller then redefines the ATA tables of L1 caches (potentially, all of them), thus redistributing the memory resources in the system.

The biggest question here is how we define the time instant at which the memory controllers must take action to redistribute resources. In this paper we will call this instant in the execution a *redistribution point*. Intuitively, this point must not happen too soon along the overall system execution, because the clustering mechanism needs values that characterize the memory demands for each application and are statistically significant. On the other hand, by doing it too late in the execution time, the pattern will be well defined, but it may be too late to overcome the overhead caused by the redistribution itself. Therefore, some trade-off should be considered in this approach.

By giving more L2 memory banks to some applications, there is no alternative but to take these resources away from other applications. The hope is to efficiently take the correct amount of L2 caches from applications that need them less and give them to applications that need them most. The aspect to be explored is which L2 caches shall be taken away at the moment of the redistribution.

The policy used in this case is to gather new resources that are closer to the resources originally allocated to the application. In this policy, it is reasonable for the applications to exchange resources placed on the edge of their clusters. That condition has the goal of keeping the resources close together, thus avoiding cluster fragmentation. The master memory controller starts by assigning the new resources to the cluster with more memory needs. Next, this process is repeated with the second cluster with more memory needs. This process goes on, with each cluster taking turns on the resource assignment, until the number of resources to be redistributed is reached.

4 Experimental Setup and Results

The experiments consider four applications: a matrix multiplication, a motion estimation algorithm, a Mergesort algorithm, and a JPEG encoder.

Considering the data inputs for the applications described above, Fig. 5 represents their communication workload regarding different numbers of processors. Based on this chart, it is expected that the Motion Estimation algorithm will generate a larger amount of data exchanges. This shows that the benchmarks used here are capable of generating significant amount of communication traffic. This would make an efficient memory hierarchy even more required in order to avoid unnecessary memory requests.

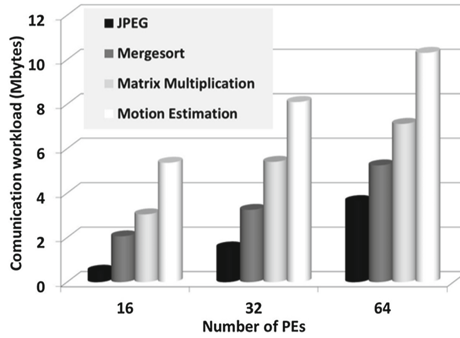


Fig. 5. Communication workload.

Experiments evaluate two characteristics: performance, measured by the total execution time of each application, and the overall dynamic energy spent, including processors, network, and memories. For the energy of the processors, a cycle-accurate power simulator [4] is used. For the network (including buffers, arbiter, crossbar, and links), the Orion library [5] is applied, and for the memory and caches the Cacti tool [6] is used.

All experiments were performed in a SystemC cycle-accurate MPSoC virtual platform that can instantiate architectural components as illustrated in Fig. 2. Due to their smaller size, this virtual platform uses MIPS processors as cores implementing MIPS I ISA. The configuration used in all experiments in this platform is presented in Table 1.

The remainder of this section presents results regarding the memory clustering experiments using the Pre-defined and On-demand Distribution policies. In this second policy the experiments were performed using four distinct redistribution points. Goal of these experiments is to find the best moment during the execution of applications to perform a memory redistribution, in a way that, after this point, the system can have an overall performance boost. The ideal time point at which the redistribution of the memory subsystem must be performed depends on each scenario, considering number of cores, number of L2 caches, execution time of all applications, etc. Since we are dealing with different MPSoC sizes, the overall execution time can vary drastically. Therefore, the

Table 1. Configuration of the virtual platform.

Number of processors	16; 32; 64
Number of L2 caches	16; 32; 64
Total number of initial tasks per application	32
L1 cache size	8192 bytes
Block size	32 bytes

redistribution points for the four experiments were defined at 10%, 25%, 50% and 75% of the overall execution time in each case, such that we can explore the impact of different values for this parameter.

Figure 6 presents the performance and energy results using the Pre-defined Distribution. One can notice that the applications with higher input data (Motion Estimation and Matrix Multiplication) have large benefits, as opposed to the JPEG and Mergesort applications. These last two applications actually present a drop in performance, down to 22%. However, there has been an increase in the overall system performance (i.e. the amount of time to execute all applications) of 34% in the best case. As for the energy results, they present a very similar pattern when compared to the performance results, reaching up to 40% of savings.

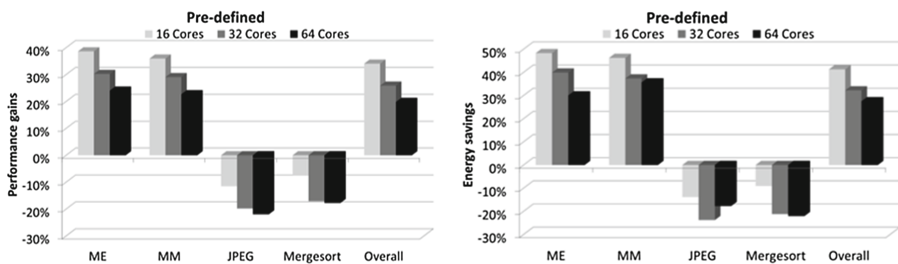


Fig. 6. Pre-defined distribution performance and energy results.

These performance and energy improvements happen because the larger applications have higher impact on the system. Obviously, due to the overhead discussed previously, these numbers are virtually impossible to achieve in practice. However, they give a good basis for understanding how good this approach can be.

Table 2 presents the results of the On-demand Distribution policy considering redistribution points of 10%, 25%, 50% and 75% of system execution time. By the results of overall performance it is possible to see that the best results occur when a redistribution point of 50% is used. This means that the execution at this point is enough to obtain a reasonable distribution of memory resources in the system. When this redistribution point occurs too early, as in the case of 10%, it almost does not affect the performance of the system. This seems to be due to the fact that this short amount of execution time is not enough to characterize memory demands of all applications correctly. Experiments with later redistribution points do not present the best results as well. There are two explanations for this behavior. First, when leaving the redistribution to occur that late in the system, the applications that needed more memory resources were not prioritized for the most part of their execution. This would have a considerable impact on the overall performance. Second, at 75% of execution time, there is not much time left to compensate the overhead caused by the memory redistribution itself.

Table 2. Performance results for on-demand distribution.

Redistrib. pts	10%			25%			50%			75%		
No. of cores	16	32	64	16	32	64	16	32	64	16	32	64
Motion Est.	3%	2%	1%	18%	12%	7%	23%	20%	16%	−5%	−3%	−2%
Matrix Multipl.	3%	2%	2%	11%	7%	5%	17%	14%	10%	−3%	−2%	−1%
JPEG	−4%	−5%	−6%	−3%	−8%	−10%	−9%	−14%	−20%	−2%	−1%	−1%
Mergesort	−2%	−3%	−5%	−3%	−6%	−7%	−2%	−5%	−8%	−3%	−2%	−1%
Overall	1.2%	0.1%	−0.6%	12%	6%	3%	18%	14%	11%	−3.5%	−2.3%	−1.4%

Table 3 presents the energy results for the On-demand Distribution. As in the case of the Pre-defined Distribution, the energy results follow a similar pattern when compared to the performance results. There are slightly higher energy savings than performance gains, probably due to the higher energy costs of accessing the external memory if com-pared to the time access penalty. Conversely, the penalty using a redistribution point at 75% of execution is higher.

Table 3. Energy results for on-demand distribution.

Redistrib. pts	10%			25%			50%			75%		
No. of cores	16	32	64	16	32	64	16	32	64	16	32	64
Motion Est.	4%	1%	1%	20%	19%	8%	25%	23%	18%	−7%	−4%	−3%
Matrix Multipl.	5%	3%	2%	12%	7%	7%	21%	19%	12%	−6%	−4%	−2%
JPEG	−5%	−6%	−7%	−3%	−8%	−9%	−9%	−15%	−22%	−3%	−2%	−3%
Mergesort	−2%	−4%	−6%	−4%	−7%	−6%	−3%	−8%	−10%	−4%	−3%	−1%
Overall	2%	0.6%	−0.2%	13%	9%	4%	22%	18%	13%	−5.6%	−3.7%	−2.3%

5 Related Work

Even though no approaches using dynamic cluster-based approaches for managing memory resources were found in the literature, some works propose resource management techniques that could be extended to the memory subsystem. In this section we present some of these works.

Qureshi and Patt [7] present a redistribution mechanism in an L1 private/L2 shared cache architecture. In this mechanism, the evicted lines from one cache (called spiller) can be allocated in another cache (called receiver) in the system through a snooping mechanism. This is an approach with the goal of extending the size for one cache (the spiller) and to make the data stay inside the chip longer. This approach considers a single address space and several threads from the same application while ours is a global approach.

Recent works propose a paradigm of multicore programming called Invasive Computing [8–10]. The proposal is to take advantage of a large many-core processor in the best possible way. The idea leverages on malleable applications where the degree of parallelism can change on the fly. At the application level, the programmer uses functions that trigger the invasion process. When an application sends an invade request, a distributed resource manager evaluates the

request based on the amount of resources required and the estimated speed-up per core. This Invasive Computing works at different levels of abstraction, and, therefore, the programmer must have some notion of the methodology. In the case of the Memory Clustering approach proposed in this paper, all mechanisms are transparent to the programmer.

6 Conclusion

In this paper we introduce the con-cept of Memory Clustering. Since some applications have more memory needs than others, the key idea of this mechanism is to reserve more memory resources (in fact L2 caches) for the more time-consuming applications, taking them away from other applications that do not need them as much. Overall results presented show that, depending on the time point where the redistribution of memory resources occurs, there is room for performance and energy gains for the system as a whole.

In the future, we intend to investigate other methods to determine the right moment for the memory redistribution. In the experiments presented here, the best time points for the redistribution (25% and 50% of the overall system execution time) have been identified for specific experiments. Therefore, in a situation where there is no previous knowledge of the applications, the best redistribution point can be different from those values. In the future, we will investigate further the cache misses before and after the redistribution points, in order to better characterize causes of the initial performance loss and effects of the memory redistribution.

References

1. Jaleel, A.: Memory characterization of workloads using instrumentation-driven simulation - a pin-based memory characterization of the SPEC CPU2000 and SPEC CPU2006 benchmark suites. Intel Corporation, VSSAD. Technical report (2007)
2. Girão, G., Santini, T., Wagner, F.R.: Exploring resource mapping policies for dynamic clustering on NoC-based MPSoCs. In: Design, Automation Test in Europe Conference Exhibition (DATE), pp. 681–684. IEEE Press, New York (2013)
3. Girão, G., Santini, T., Wagner, F.R.: Cache coherency communication cost in a NoC-based MPSoC platform. In: Proceedings of the 20th Annual Conference on Integrated Circuits and Systems Design, pp. 288–293. ACM, New York (2007)
4. Chen, B., Nedelchev, I.: Power compiler: a gate-level power optimization and synthesis system. In: IEEE International Conference on Computer Design, pp. 74–79. IEEE Press, New York (1997)
5. Kahng, A.B., Bin, L., Peh, L.-S., Samadi, K.: ORION 2.0: a power-area simulator for interconnection networks. IEEE Trans. VLSI **20**, 191–196 (2011)
6. Wilton, S.J.E., Jouppi, N.P.: CACTI: an enhanced cache access and cycle time model. IEEE J. Solid-State Circuits **31**, 677–688 (2002)
7. Qureshi, M.K., Patt, Y.N.: Utility-based partitioning of shared caches. In: 39th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 423–432. IEEE Press, New York (2006)

8. Teich, J., et al.: Invasive computing: an overview. In: Hübner, M., Becker, J. (eds.) Multiprocessor System-on-Chip Hardware Design and Tool Integration, pp. 241–268. Springer, Heidelberg (2011). https://doi.org/10.1007/978-1-4419-6460-1_11
9. Henkel, J., et al.: Invasive manycore architectures. In: 17th Asia and South Pacific Design Automation Conference, pp. 193–200. IEEE Press, New York (2012)
10. Kobbe, S., et al.: DistRM: distributed resource management for on-chip many-core systems. In: 9th International Conference on Hardware/Software Codesign and System Synthesis, pp. 119–128. IEEE Press, New York (2011)