Chair for Integrated Systems
Department of Electrical and Computer Engineering
Technical University of Munich

Simulator Support for Dynamic Data Migration

Master thesis

Author: Iffat Brekhna

Advisor: Sven Rheindt M.Sc
Supervisor: Prof. Dr. sc. techn. Andreas Herkersdorf

Submission date: May 31, 2018

Abstract

An abstract is defined as an abbreviated accurate representation of the contents of a document. – American National Standards Institute (ANSI)

Contents

List of Figures **5**

List of Tables **6**

1 Introduction **7**
1.1 Motivation . . . 7
1.2 Problem . . . 7
1.3 Goals of Thesis . . . 8
1.4 Drawbacks . . . 8
1.5 Approach . . . 9
1.6 Outline . . . 9

2 Background and Related Work **10**
2.1 Basic Concepts . . . 10
2.1.1 Tile . . . 10
2.2 Related Work . . . 11
2.2.1 Data Placement/Migration on Caches . . . 11
2.2.2 Task/Thread Placement . . . 12
2.2.3 Data and Thread Migration . . . 13
2.2.4 Data Placement on TLM . . . 13

3 System Architecture **14**
3.1 Local Accesses . . . 14
3.2 Remote Acesses . . . 14

4 Approach **16**
4.1 Overview . . . 16
4.1.1 Cache_Stats Module . . . 17
4.2 garbage . . . 17
4.3 Concept . . . 18

5 Evaluation **23**

6 Conclusion and Outlook **24**

Bibliography **25**

List of Figures

Figure 1.1 Invasic Architecture [12] 8

Figure 2.1 A Tile 11

Figure 3.1 A tile showing local accesses to a TLM 15
Figure 3.2 A tile showing remote accesses to a TLM 15

Figure 4.1 Diagram showing overview of the modules used 16
Figure 4.2 Diagram showing where the trace file is placed 17
Figure 4.3 Diagram showing approach of the solution 18
Figure 4.4 Flowchart showing process inside cache_stats module 19
Figure 4.5 Flowchart showing how the free address space in TLM is calculated 20
Figure 4.6 Flowchart showing process inside central TLM_STATS module . . 21
Figure 4.7 module connections 22

List of Tables

1 Introduction

This chapter serves as an introduction to the thesis. It explains the motivation for undertaking this work and the approach and concepts used in building the simulator support for dynamic data placement. Finally, the outline of the work is presented.

1.1 Motivation

Current research in semi-conductor industry is towards developing a single chip multi-tile multi-core processor. Hence, parallel programming is experiencing a rapid growth with the advent of architectures like the Invasic architecture as shown in Figure 1.1. Because of multiple tiles and cores on one chip these processors deal with data processing at a high scale and complexity. Therefore, the bottleneck have shifted from computational complexities to data management capacities.

Since modern, scalable multiprocessor system-on-chip (MPSoC) platforms have Non-Uniform Memory Access (NUMA) properties, application performance is highly influenced by data-to-task locality. The goal is to bring tasks and data closer together to increase overall performance. This is a twofold and complementary problem consisting of data and or task migration. In this thesis, we will look into data placement and see how it improves the performance of the MPSoC.

We propose a dynamic data placement (DDP) scheme in which the data is migrated dynamically at run time from one Tile Local Memory (TLM) to another TLM if the need arises. This is the major differentiating factor of our approach, managing data placement at run time rather than at compile time.

1.2 Problem

Formerly, data placement was done statically at compile time. It is not a efficient solution since the best data location is found after running the application (trace file) once and studying its memory access pattern. Then the data is placed according to the memory access pattern and the application runs again. This is not a very realistic approach since the application has to run twice and in real life you don't know how an application will behave in the future.

Also, in static data placement once the data is placed on TLM's that placement is fixed, you cannot change it even if the placement is having negative effects on the performance of the processor. You have to restart the application if you want to change the data placement.

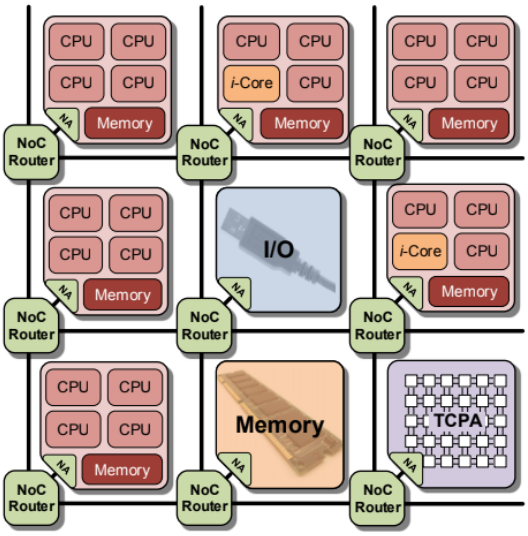

Figure 1.1: Invasic Architecture [12]

1.3 Goals of Thesis

The goal of this thesis is to design, implement and evaluate a dynamic data placement technique for memory management at run time. The system will evaluate itself and find the best data placement to improve it's performance.
The outcome will be a system which does not need external support to find the best placement for its memory but rather it will adapt itself to place the data at the best location which will in turn improve the performance.

1.4 Drawbacks

The drawback of this thesis could be increased traffic in the processor since quite amount of data is sent back and forth in order to find the best data placement. Also, the modules used in this technique will need memory and that memory might not scale with increasing number of cores.

1.5 Approach

The steps followed to develop the Simulator Support for Dynamic Data Migration on a Distributed Shared Memory System ... are as follows:

- Understanding the Idea: In this step, the purpose is to understand why we need data migration in the first place and how data placement is done on a distributed shared memory before the application runs. We look into other means of bringing the data close to the processor as well.
- Literature Review: Find and read relevant work that has already been done for bringing the data and process/task together. Choose one approach on how to bring data and task together and find relevant ideas to understanding the concept better.
- Design: Here we decide which technologies to use and how to design our system for optimal results. We want a design thats easy to change, extend, optimize and is scalable. Also, we decide how we will evaluate our system eventually and what metrics we will use in result gathering.
- Implementation: Implement the design of the solution.
- Evaluation and Testing: Compare the results of thesis with static data placement results. –complete this after you get results iffat
- Writing Report: Compose a document that explains the system in detail and depict the results obtained from using this system/thesis??.

1.6 Outline

The work is structured as follows. In Chapter 1, a brief introduction of the problem is given along with the motivation to solve it and then a brief overview of the solution is given.

In Chapter 2, the basic concepts of data migration and management along with the work already done related to this thesis is given.

In Chapter 3, the solution/approach is explained in detail and it is shown how the solution has been implemented.

In Chapter 4, the results are presented. It shows how the performance has changed with the proposed solution implementation.

In Chapter 5 a summary is given and some suggestions for future works in this domain.

2 Background and Related Work

In this chapter the necessary background information is introduced in order to understand the thesis. Moreover, we will discuss the related work in this domain of research.

2.1 Basic Concepts

2.1.1 Tile

Figure 2.1 shows a single tile. You can see in the figure that it composes of four CPU cores, L1 caches for every core, L2 cache which is shared between all the cores and a TLM memory which is also shared by all the cores.

CPU core

A CPU core is the basic processing unit that receives instructions(from the user or application) and performs calculations based on those instructions. A processor can have a single core or multiple cores.

TLM

TLM stands for Tile Local Memory. Each tile has its own TLM which is shared among all the cores of the tile [6], [12]. This memory is cachable by the L1 caches of all the cores in the tile it sits on and by the L2 cache of any other tile. The TLM from one tile can be accessed by the core of another tile.

Cache

Cache is a temporary storage space which is made up of high-speed static RAM (sRAM). It stores information which has been recently accessed so that it can be quickly accessed at a later time. It operates on the principle that most programs access information or data over and over again so by having data in the SRAM the CPU does not access the slow DRAM again and again. A cache hit occurs when the processor core accesses data that is already present in the cache whereas a cache

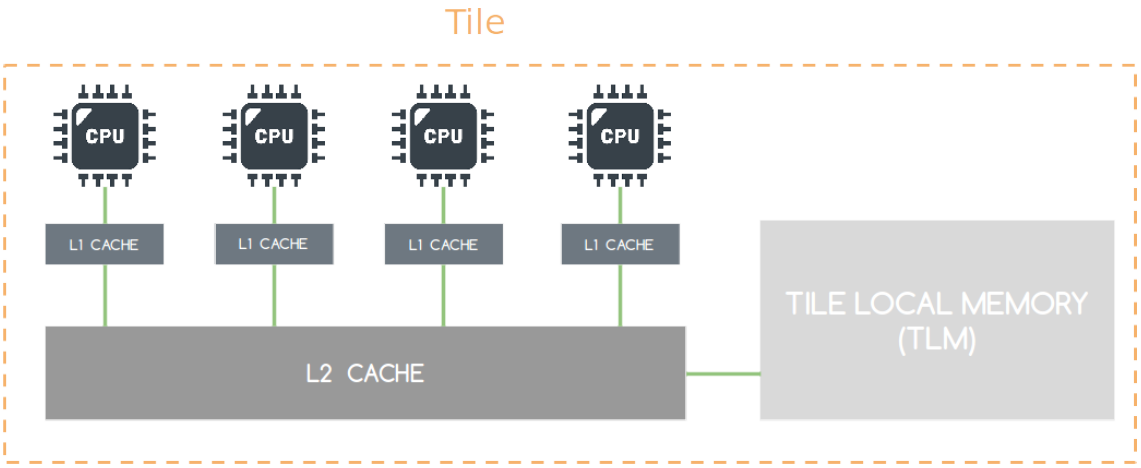

Figure 2.1: A Tile

miss occurs when the data is not present in the cache and has to be fetched from the TLM or the main memory to the cache [1]. In our architecture we have two levels of caches:

- L1 Cache: Level 1 cache (L1 Cache) is the cache right next to the core and is the smallest in size. It is not shared with any other core i.e it is a private cache.
- L2 Cache:Level 2 cache (L2 Cache) is away from the processor and is larger in size than the L1 cache. It is shared between all the cores in a tile. In our scenario, L2 cache is the Last Level Cache (LLC) in the system.

2.2 Related Work

2.2.1 Data Placement/Migration on Caches

A great amount of work have been done on data-placement in the shared last level cache in order to reduce the distance of data from the core requesting the data and to take care of load balancing across the chip.

In static data placement [2], [5] the whole address space is divided into subsets and every subset is mapped to a LLC slice regardless of the location of the requesting core which leads to unnecessary on-chip traffic. Its advantage is that it evenly distributes the data among the available LLC slices and reduces off-chip accesses. In dynamic data placement [2], [14], [10] the data blocks are placed such as to reduce the distance between the data block's home node and the core requesting it. This eliminates the unnecessary on-chip traffic. It requires a lookup mechanism to locate

the dynamically selected home node for each data block. In reactive data placement data is classified as private or shared using the operating systems page tables at page granularity [10], [11]. Because all placement is performed at page granularity level there is load imbalance as some LLC slices might have higher accesses compared to others. This load imbalance leads to hot-spots [11].

There is a hybrid data placement [11] which combines the best features of static and dynamic data placement techniques. It optimizes data locality and also takes care of load balancing of the shared data. Hybrid data placement differs from Reactive data placement in regard to allocation of shared data among the cores i.e in Hybrid data placement, data is also classified as private or shared using the operating systems page tables but when a page is classified as shared (in hybrid data placement) it is allocated to a cluster of LLC slices and within this cluster the page is statically interleaved at the granularity of cache lines [11]. This balances the load among the LLC slices.

2.2.2 Task/Thread Placement

Placing threads that share data on the same core improves performance [4]. However, finding the optimal mapping between threads and cores is a NP-hard problem [7] and cannot be scaled. One way to solve this problem is by monitoring the data accesses to determine the interaction between threads and the demands on cache memory [8]. In [8] a mechanism is there to transform the number of memory accesses from different threads to communication patterns and used these patterns to place the threads that share data on cores that share levels of cache. They generate a communication matrix using the number of accesses to the same memory location by two threads and then maps the threads with highest communication to the same core. The disadvantage of this method is that generating the communication matrix through simulation is slow and they propose the application vendor provides this matrix with the application.

In [3] a thread scheduling mechanism is proposed which uses the performance monitoring unit (PMU) with integrated hardware performance counters (HPCs) available in today's processors to automatically re-cluster threads online. Using HPSs they monitor the stall breakdowns to check if cross chip communication is the reason for the stalls. If that is so, they detect the sharing pattern between the threads using the data sampling feature of the PMU. For every thread they maintain a summary vector called the shMap which holds the signature of data regions accessed by the thread which resulted in cross-chip communication. These shMaps are analyzed i.e threads with high degree of sharing will have similar shMaps and will be placed to the same cluster. The OS then migrates the threads with higher sharing to the same

cluster and place them as close as possible [3].

2.2.3 Data and Thread Migration

In [9] a mechanism called CDCS is presented which using a combination of hardware and software techniques jointly places threads and data in multi-cores with distributed shared caches. CDCS takes a multi-step approach to solve the various interdependencies. It places data first and then places threads such that the threads are close to the center of mass of their data. Then using the thread placement it again re-place the data and once again for this data it re-places the threads to get a optimum placement. This technique improves performance and energy efficiency for both thread clustering and NUCA techniques [9].

2.2.4 Data Placement on TLM

Static Data Placement on TLM

In static data placement the application is run twice. The first time it runs in order to find the best placement for the data and then for the second time it runs over the data placement done in the previous step. The disadvantages of this approach are that the data placement is fixed i.e even if some data block is becoming a bottleneck in performance the processor cannot migrate the memory block at rum time. It can only be changed by resetting the system.

Dynamic Data Migration on TLM

In [13] the authors have proposed a dynamic page migration scheme for a multiprocessor architecture using point-to-point interconnects with a distributed global memory. They use the *pivot* mechanism to regulate the dynamic migration of pages by keeping track of the access pattern to every local page in every distributed memory module. If the access pattern is unbalanced then the page pivots to the nearest neighbor in the direction which caused the unbalanced access pattern.

In acquiring the results the authors assumed two sets of conditions:

- infinite memory space model i.e it is assumed that the destination memory module always has free space
- finite memory space model i.e a page is only allowed to migrate if its destination memory module has free space

Ask Sven whether to put their results graphs here or not!

3 System Architecture

As mentioned in the previous chapter, figure 2.1 depicts the inside of one tile. We have multiple such tiles in our processor hence the name multi-tile multi-core processor architecture.

3.1 Local TLM Accesses

Figure 3.1 depicts a scenario where a core is accessing its own tiles TLM which makes it a local TLM access.

3.2 Remote TLM Accesses

Figure 3.2 depicts a scenario where a core is accessing another tiles TLM which makes it a remote TLM access.

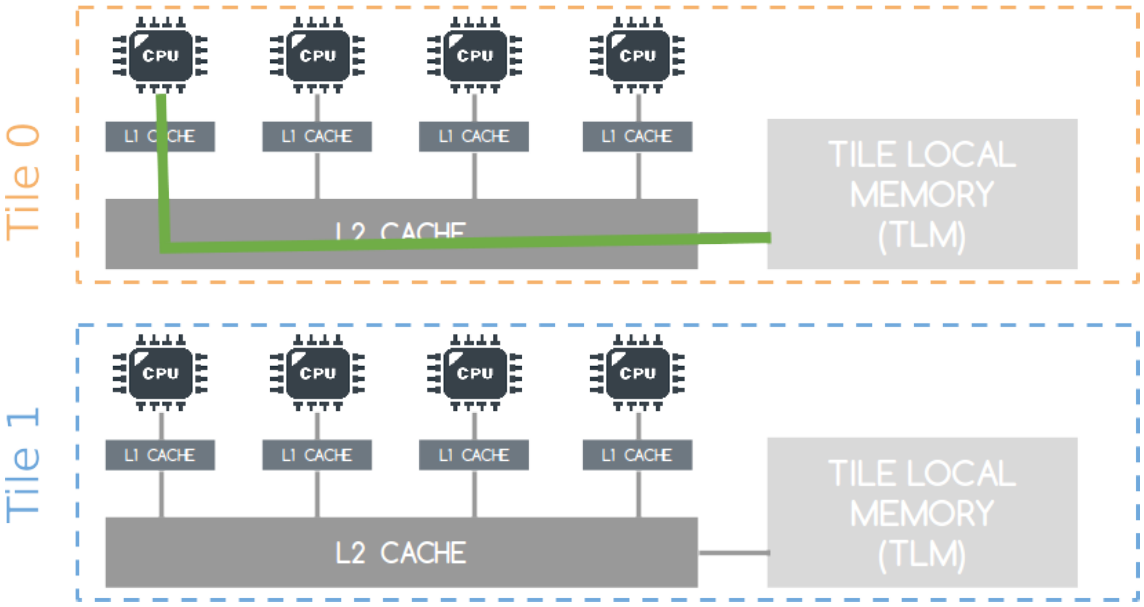

Figure 3.1: A tile showing local accesses to a TLM

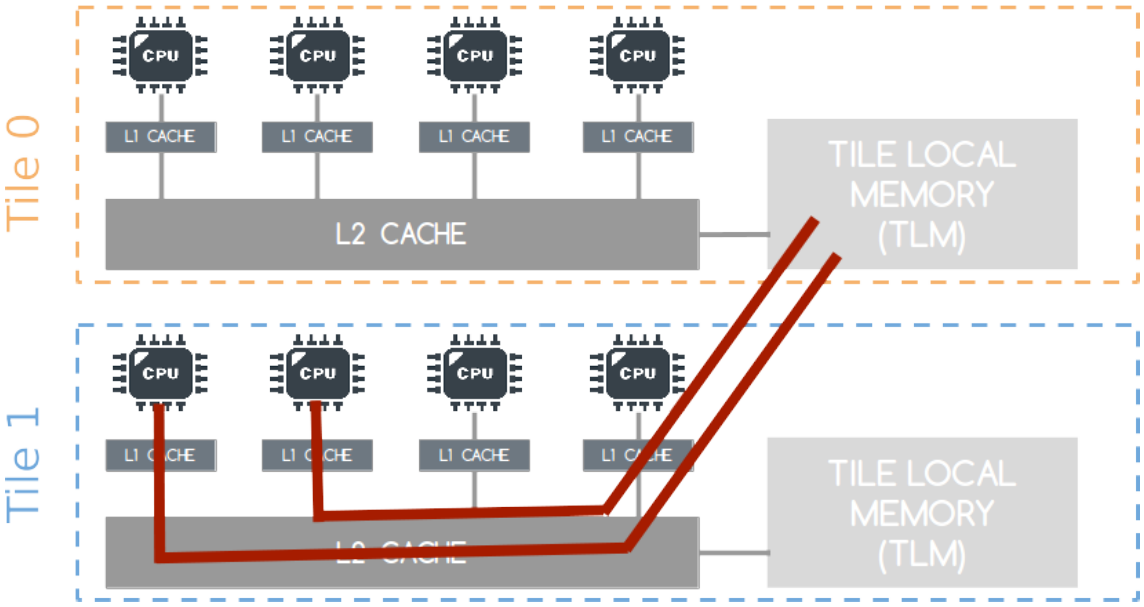

Figure 3.2: A tile showing remote accesses to a TLM

4 Approach

4.1 Overview

Figure 4.1 shows the overview of the modules involved in dynamic migration scheme. Every cache module is connected to a cache_stats module. All these cache_stats modules and the TLM_MEM modules report to the central TLM Stats module at every given time interval ($T_{interval}$). The central TLM Stats module does evaluation of this data and triggers migration if needed.

Also, there is a vector address table which sits between the trace file and the CPU's as shown in Figure 4.2. This table contains the address translation of all the addresses from the DRAM to the TLM. At start all TLM's are empty which means every instruction has to access data from the DRAM. The vector address table is updated if the DRAM is accessed or if migration is triggered by the central TLM Stats module and that migration takes place.

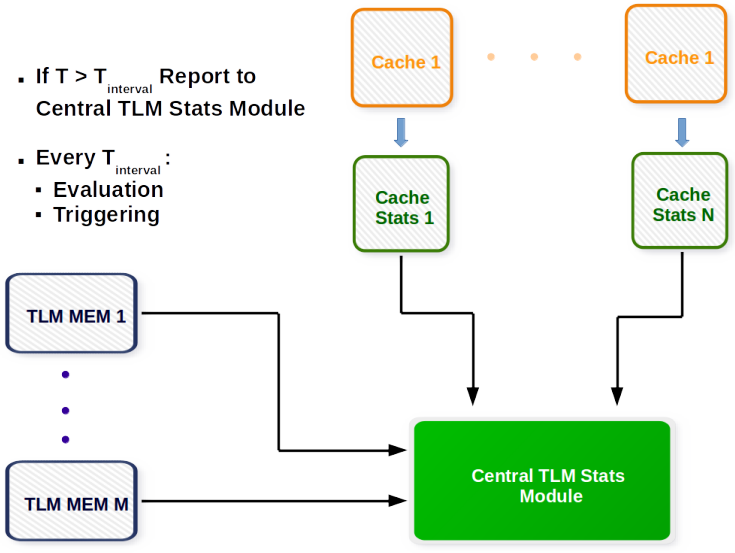

Figure 4.1: Diagram showing overview of the modules used

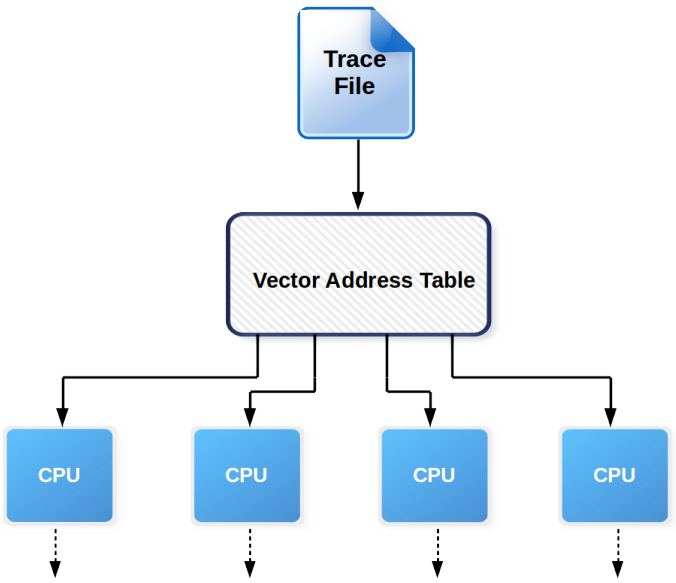

Figure 4.2: Diagram showing where the trace file is placed

4.1.1 Cache_Stats Module

The cache_Stats module is connected to the L1 and L2 Cache modules and is continuously getting updates from them regarding cache hits, misses, evictions etc per cache line. This module bring all the data together and at the end of the simulation prints the compulsory misses,conflict misses, evictions, local misses, remote misses, local hits, remote hits for all the caches. This module also calculates the number of local and remote access to the TLM block by using all the metrics mentioned above.

4.2 garbage

Figure 1.1 show a multi-tile multi-core processor architecture. Each tile has four Central Processing Units (CPU's) and a Tile Local Memory (TLM) which can be accessed by other tiles CPU's. Hence, there are local and remote accesses to the TLM. We need dynamic data placement in order to reduce these remote accesses at run time so that the distance of data from the core requesting the data is localized(minimized). Previously data was placed on the TLM's statically at compile time. However, this is not a very efficient or realistic way since in real life we don't know how an application or program will behave in the future. A more realistic way would be to dynamically place the data by examining the behavior of the processor for a given time interval and taking data placement decisions on that basis.

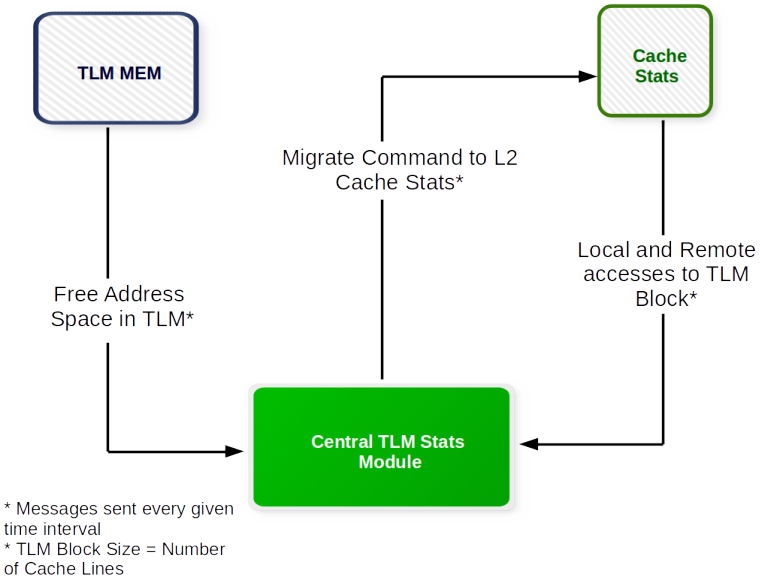

Figure 4.3: Diagram showing approach of the solution

4.3 Concept

Fig. 4.3 shows the concept of the dynamic migration scheme. It shows what messages/data are exchanged between the different modules explained/introduced above. At every given time interval ($T_{interval}$) the cache_stats module sends the number of local and remote accesses of a TLM block to the central TLM Stats module and the TLM_MEM module sends its free address space to the central TLM stats module. This block size is equal to a number of cache lines and can be varied. The central TLM Stats module evaluates this data and sends a migrate command to the L2 Cache_stats module if migration shall be done.

Fig. 4.4 shows the how the metric for local access and remote access is calculated in the cache_stats module. The TLM which is being accessed is compared with the current tile. If the two values are equal it means it is a request to the TLM of the same tile which means its a local access. If the two values are different it means it is a request for another tile's TLM and it is checked whether there is a L1 cache hit. If there is a L1 cache hit it means it is a local access and if it is a miss it is checked whether it is a L2 cache hit. If it is a hit it means it is a local access and if it is a miss it means it is a remote access.

Figure 4.5 shows how the free address space is calculated in the TLM_MEM module. The vector address table is checked first and if it is empty it means all TLM's are empty. If the table is not empty and there is free space in a TLM, the starting and

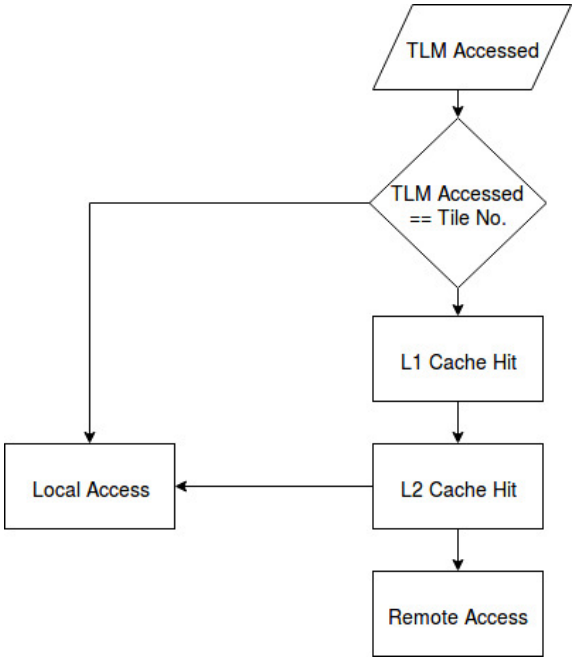

Figure 4.4: Flowchart showing process inside cache_stats module

ending address of free space is extracted. If there is so free space in a TLM that is sent to the central TLM_Stats module.

Figure 4.6 shows the algorithm for determining when migration shall take place and which TLM block to migrate. First it is determined which node to migrate the TLM block to. This decision is based on the local and remote accesses to the specific TLM block. The tile with the maximum remote accesses to the TLM block is found and if these remote accesses are greater than local accesses to that TLM block it means it has to be migrated to the tile with the highest remote accesses. Then it is determined whether there is free space in the TLM of the tile to which the TLM block is to be migrated. If there is free space, a migration command is send to L2 Cache_—Stats module. The migrate command is split to two commands, first reading data from the location from where data has to be migrated and then writing data at the new location. Once, the data is read a invalidation command is sent for that TLM Block/Cache Line(s) and the vector address table is updated. However, if there is no free space the block with the least number of local accesses is found in the TLM and that metric is compared with the remote accesses of the TLM block to be migrated. If the remote accesses metric is higher than the local accesses of the block with least number of local accesses a migration command is send to the L2 Cache_—Stats module for this block i.e this block is migrated back to the DDR and free space is made for the incoming TLM block. Now with free space in the TLM a migration command is send to the L2 Cache_—Stats module

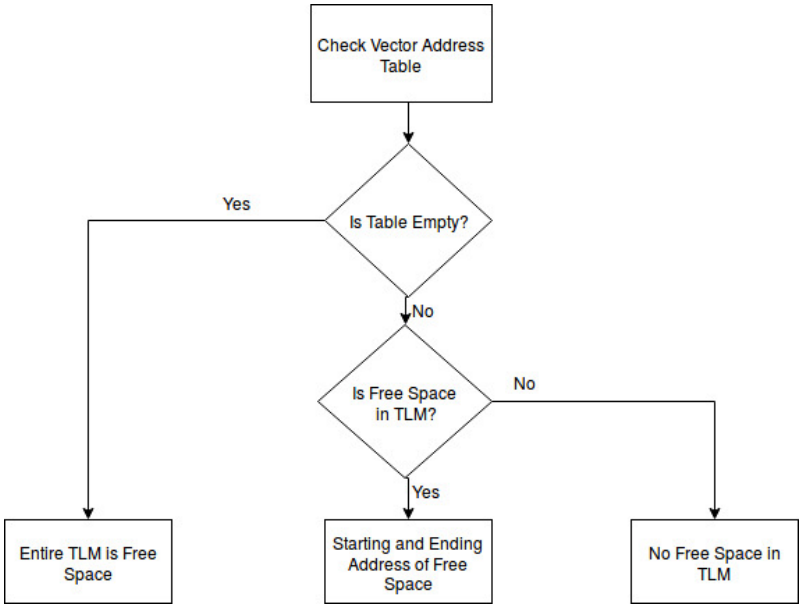

Figure 4.5: Flowchart showing how the free address space in TLM is calculated for the block to be migrated.

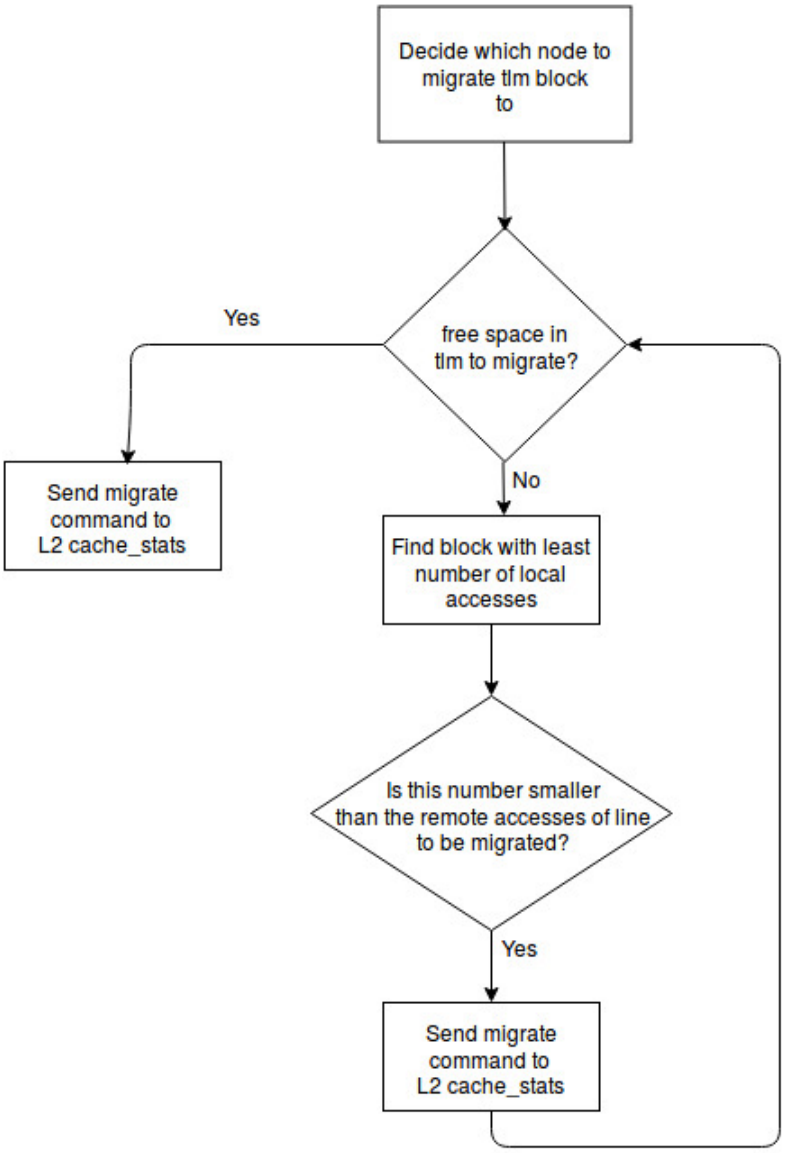

Figure 4.6: Flowchart showing process inside central TLM_STATS module

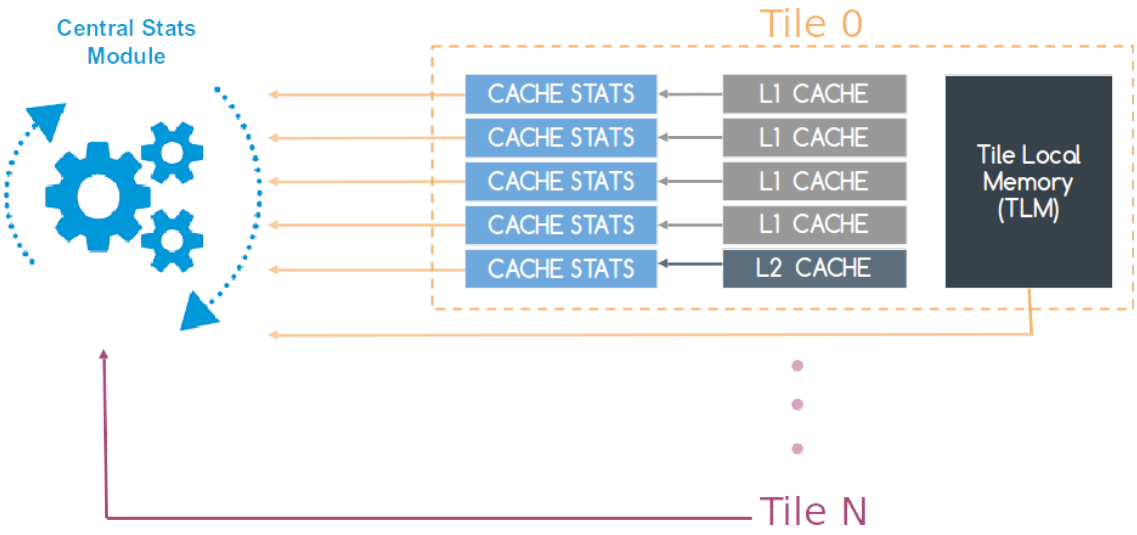

Figure 4.7: module connections

5 Evaluation

6 Conclusion and Outlook

Bibliography

[1] Iffat Brekhna. Cache coherency and data/task placement, 2016.

[2] D. Burger C. Kim and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. International Conference on Architectural Support for Programming Languages and Operating Systems, 2002.

[3] R. Azimi D. Tam and M. Stumm. Thread clustering: Sharing-aware scheduling on smp-cmp-smt multiprocessors. Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, 2007.

[4] P. Navaux E. Rodrigues, F. Madruga and J. Panetta. Multi-core aware process mapping and its impact on communication overhead of parallel applications. IEEE Symposium on Computers and Communications, 2009.

[5] C. Fensch and M. Cintra. An os-based alternative to full hardware coherence on tiled cmps. International Symposium on High Performance Computer Architecture, 2008.

[6] Ankit Kalbande. Hardware support for configurable cache-coherency in tiled many-core architectures, 2015.

[7] R. Koppler. Geometry-aided rectilinear partitioning of unstructured meshes. Lecture Notes in Computer Science, 1999.

[8] H.U. Heiss F.L Madrunga E. R. Rodrigues M.A.Y. Alves M. Diener, J. Schneider and P.O.A Navaux. Evaluating thread placement based on memory access patterns for multi-core processors. 12th IEEE International Conference on High Performance Computing and Communications, 2010.

[9] P. Tsai N. Beckmann and D. Sanchez. Scaling distributed cache hierarchies through computation and data co-scheduling. Proceedings of the 21st International Symposium on High Performance Computer Architecture (HPCA), 2015.

[10] B. Falsafi N. Hardavellas, M. Ferdman and A. Ailamaki. Reactive nuca: Near-optimal block placement and replication in distributed caches. International Symposium on Computer Architecture, 2009.

[11] F. Hijaz Q. Shi and O. Khan. Towards efficient dynamic data placement in noc-based multicores. IEEE 31st International Conference on Computer Design (ICCD), 2013.

[12] Sven Rheindt. inetworkadapter documenation, 2016.

[13] Christoph Scheurich and Michel Dubois. Dynamic page migration in multiprocessors with distributed global memory. IEEE Transactions on Computers, 1989.

[14] M. Zhang and K. Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. International Symposium on Computer Architecture, 2005.

Confirmation

Herewith I, Iffat Brekhna, confirm that I independently prepared this work. No further references or auxiliary means except those declared in this document have been used.

Munich, May 31, 2018

.....
Iffat Brekhna