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 21, 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 **6**

List of Tables **7**

1 Introduction **8**

1.1 Motivation 8
1.2 Problem 8
1.3 Goals of Thesis 10
1.4 Approach 10
1.5 Outline 10

2 Background and Related Work **12**

2.1 Basic Concepts 12
2.1.1 Tile 12
2.2 Related Work 13
2.2.1 Data Placement/Migration on Caches 14
2.2.2 Task/Thread Placement 14
2.2.3 Data and Thread Migration 15
2.2.4 Data Placement on TLM 15

3 Concept and System Architecture **18**

3.1 Concept 18
3.1.1 Tile Local Memory 18
3.1.2 Goal 19
3.2 Modules Directly Used in the System for Implementing Thesis 21
3.2.1 Trace File 21
3.2.2 Memory Management Unit 21
3.2.3 Cache Stats Module 21
3.2.4 Tile Local Memory Module 21
3.2.5 Central Stats Module 21
3.3 System Design 22

4 Implementation **24**

4.1 Tools and Technologies 24

4.2 Dynamic Data Migration Process Mechanism . . . 24
4.2.1 Local and Remote Accesses to a TLM Block . . . 25
4.2.2 Free Address Space in TLM . . . 25
4.2.3 Triggering Migrations . . . 28
4.3 Usability Improvement . . . 29
4.4 Limitations . . . 30

5 Experimental Setup 32
5.1 Size of Modules . . . 32
5.2 The gem5 Simulator and Trace Files . . . 32
5.3 Platform Architect MCO . . . 33
5.4 Writing Shell Script . . . 33
5.5 *Nice* Command . . . 33
5.6 Output Files . . . 33
5.6.1 Screen Log File . . . 33
5.6.2 Time Log File . . . 33
5.7 System Specifications . . . 34

6 Evaluation 35
6.1 Dynamic Data Migration vs First Touch and Most Accessed . . . 35
6.1.1 Blackscholes . . . 35
6.1.2 Swaptions . . . 35
6.1.3 Fluidanimate . . . 35
6.1.4 Canneal . . . 35
6.2 Dynamic Data Placement for Different TLM Block Size . . . 35
6.3 Dynamic Data Placement for Different Time Interval ($T_{interval}$) . . . 35

7 Conclusion and Outlook 36
7.1 Summary . . . 36
7.2 Future Work . . . 36

Bibliography 37

List of Figures

Figure 1.1 Invasic Architecture [18] 9

Figure 2.1 A Tile 13
Figure 2.2 A mesh-connected distributed global memory system with 16 processors [19]. 17

Figure 3.1 Our architecture. 19
Figure 3.2 A tile depicting local accesses to a TLM. 20
Figure 3.3 A tile depicting remote accesses to a TLM. 20
Figure 3.4 Figure illustrating the overview of the modules used. 22
Figure 3.5 Vector Address Table and Trace File Placement 23

Figure 4.1 Messages exchanged between different modules. 25
Figure 4.2 Flowchart illustrating how to determine a local and remote TLM access. 26
Figure 4.3 Flowchart depicting how the free address space in the TLM is calculated. 26
Figure 4.4 Flowchart showing how the free address space in TLM is calculated if the TLM is partially full. 27
Figure 4.5 Process to determine the TLM Block to migrate and the tile to migrate the TLM Block to. 28
Figure 4.6 Triggering Migration Commands 30

List of Tables

Table 4.1 Table showing the size of the modules 31

Table 5.1 Table showing Size of TLM, L1 Cache and L2 Cache 32

1 Introduction

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 system on chip (SoC) architectures. One example of such a processor is the Invasic architecture [6] as shown in Figure 1.1. The main idea of Invasic architecture architecture is to introduce "resource aware programming" support so that a program has the ability to explore its surroundings and dynamically spread its computations to neighboring processors [6]. 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.

Modern, scalable multiprocessor system-on-chip (MPSoC) platform's memory access time depends on where the memory is located relative to the core which accesses the memory. In other words (MPSoC) platform's have Non-Uniform Memory Access (NUMA) properties, hence 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 migration (DDM) 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 from the current research in the group; managing data placement at run time rather than at compile time.

1.2 Problem

In [1] the research in the group has been described. The authors explain how data is placed on the tile local memory. They divide the data used by an application at the granularity of cache lines and migrate it from the global memory to the local memory according to the ideal location that is derived based on the task-to-data mapping techniques. They propose two techniques on how to do this task-to-data mapping:

First Touch Policy: As the names suggest, in first touch the memory blocks are migrated to the TLM of the tile that accesses the data first [1]. The drawback of first touch policy is that data can be placed at a local memory far away from the core that accesses it frequently hence increasing the memory access time. You cannot change the location of data at run time even if it is giving performance disadvantages.

Most Accesses Policy: In most accessed the memory blocks are migrated to the TLM of the tile that accesses it the most over the complete application [1]. The drawback of the most access policy is that the evaluation is performed for a complete application's runtime so data cannot be migrated dynamically at run time but instead it is only migrated statically at compile time.

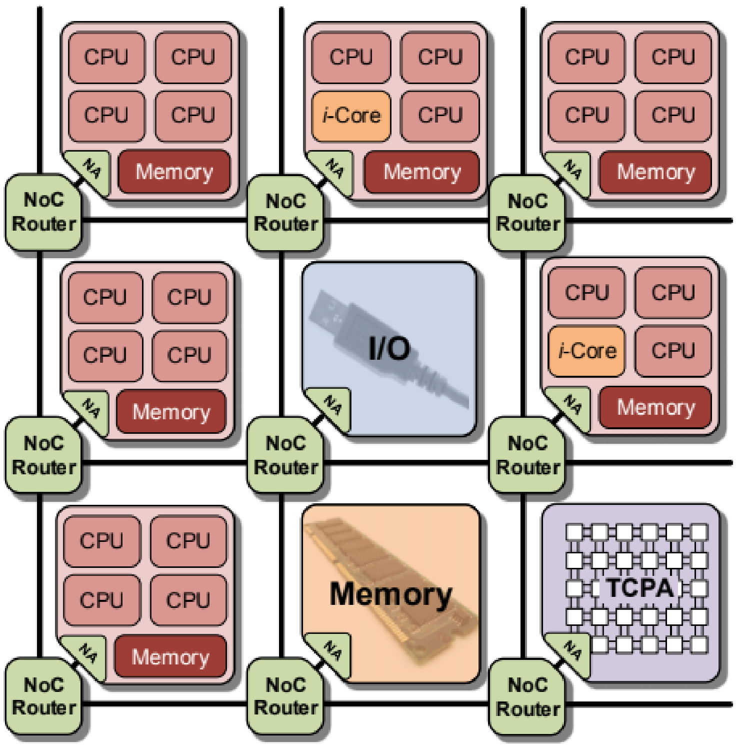

Figure 1.1: Invasic Architecture [18]

1.3 Goals of Thesis

The goal of this thesis is to design, implement and evaluate a dynamic data migration 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 Approach

The steps followed to develop the Simulator Support for Dynamic Data Migration 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 statically on a distributed shared memory system. 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 that is 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.

1.5 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 needed to understand how dynamic data migration is implemented are explained along with the work already done related to this thesis is given. In Chapter 3 the system architecture is introduced i.e the modules added for implementing dynamic data migration are introduced and explained. In Chapter 4 the implementation is explained in detail. In Chapter 5 the the programs that assisted or were used for running the simulations before gathering results are given. In Chapter 6 the results are presented. It shows how the performance has changed with the proposed solution implementation.In Chapter 7 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 Tile Local Memory (TLM) which is also shared by all the cores. It also has a Bus which connects the cores to the L2 Cache and the TLM. Each component of the tile is explained below.

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 CPU Cores of the tile [10], [18]. This memory is cachable by the L1 caches of all the CPU Cores within the tile that 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.

Bus: The bus connects the CPU Core to the L2 Cache and the TLM. Also, when another tile has to be accessed the request goes through the bus to the network adapter.

Network Adapter (NA): The network adapter provides the interface between a tile and the network connection which is providing a connection to other tiles and the DDR.

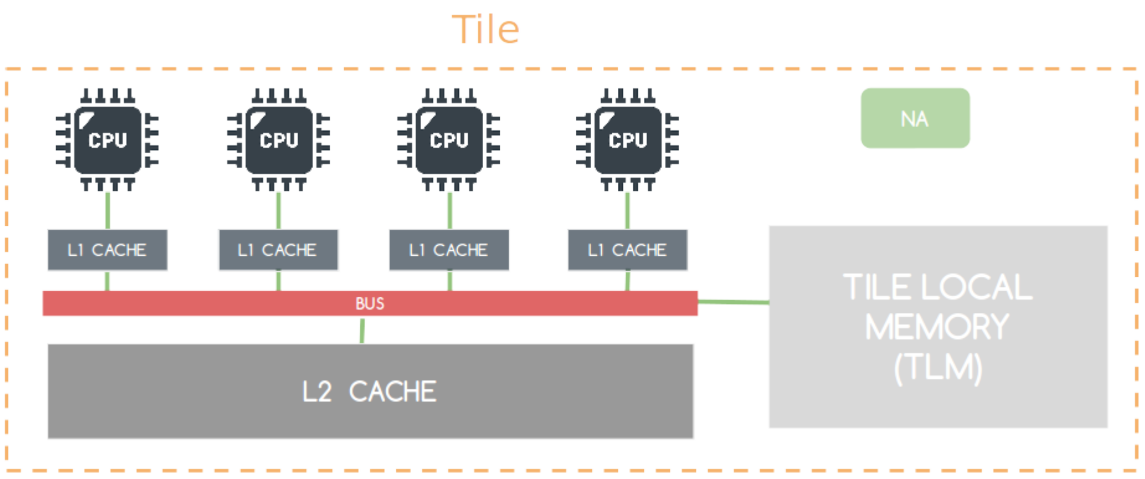

Figure 2.1: A Tile

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 DDR/DRAM again and again. A cache hit occurs when the processor core accesses data that is already present in the cache whereas a cache 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. 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

The main idea is to bring data and tasks close together in order to save execution time used in transferring data back and forth. This can be achieved in several ways. We can bring data closer to the tasks by placing it in the cache of the core accessing the data or by migrating it to the cache of the core accessing the data. We can also migrate tasks to data instead of migrating data all the time. Also, we can migrate data to the local memory near the core accessing the data. These different ideas on how it has been done are discussed in the subsections below.

2.2.1 Data Placement/Migration on Caches

This section explicitly talks about how data placement and migration has been implemented in caches. A great amount of work have been done on data-placement in the shared last level cache (LLC) 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], [8] 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], [22], [15] 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 [15], [17]. 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 [17].

There is a hybrid data placement [17] 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 [17]. 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 [7]. However, finding the optimal mapping between threads and cores is a NP-hard problem [11] 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 [12]. In [12] a mechanism is there to transform the number of memory accesses from different threads to communication patterns and use 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 [5] 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 [5].

2.2.3 Data and Thread Migration

In [14] 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 [14].

2.2.4 Data Placement on TLM

This section talks about the data placement mechanisms on the tile local memory.

Static Data Placement on TLM

In [1] the authors have implemented static data placement for the tile local memory. They divide the data used by an application at the granularity of cache lines and migrate it from the global memory to the local memory according to a ideal location that is derived based on the task-to-data mapping techniques. They propose two schemes on how to do this task-to-data mapping:

First Touch Policy: As the names suggest in First Touch Policy the memory blocks are migrated to the TLM of the tile that accesses the data first. The drawback of first touch policy is that data can be placed at a local memory far away from the core that accesses it frequently hence increasing the memory access time. You cannot change the location of data at run time even if it is giving performance disadvantages.

Most Accesses Policy: In Most Accessed Policy the memory blocks are migrated to the TLM of the tile that accesses it the most over the complete application runtime. The drawback of this policy is that the evaluation is performed for a complete application's runtime so data cannot be migrated dynamically at run time but instead it is only migrated statically at compile time.

Dynamic Data Placement and Migration on TLM

In [19], [20] the authors have proposed a dynamic page migration scheme for a multiprocessor architecture using a mesh connection with a distributed-shared global memory as shown in figure 2.2. Initially, the pages are distributed among the processors. This initial page allocation might be bad since the allocation might result in large average distances between the data and the processors. However, this bad placement can be made better by dynamically migrating the pages.

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. They use dedicated hardware for a pair of up and down counters associated with every page frame to keep track of the number of accesses made by processors in located in different rows and columns. When the absolute value of either of these two counters exceeds a certain threshold value which they called the pivot factor, the page is pivoted or moved to the nearest neighbor.

Also, to support the dynamic memory allocation, they also have a TLB (translation look-aside buffer) [4] in-order to support the translation of logical addresses into physical addresses for the pages. Since, the pages can migrate, the coherence among the TLB's in different processors must be maintained.

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

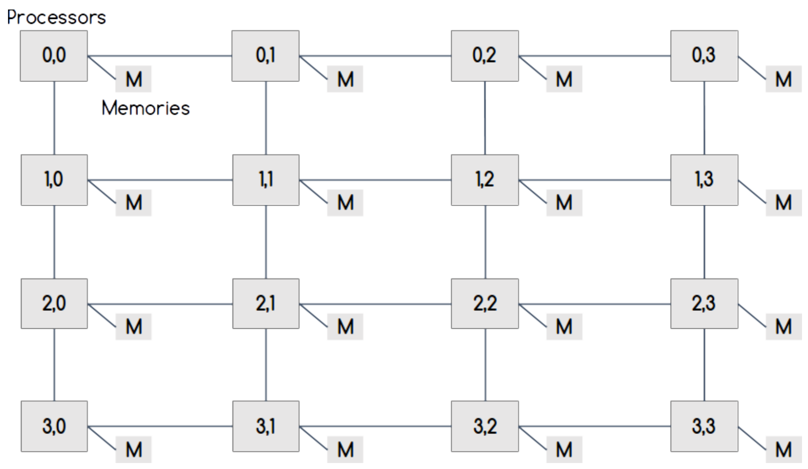

Figure 2.2: A mesh-connected distributed global memory system with 16 processors [19].

memory module has free space

The authors compared a parameter called *hop* for dynamic page migration turned on and off. A *Hop* is the traversal of an access through a single processor-to-processor link [19]. A access by a processor to a direct neighbor memory is counted as a single hop. In all the cases the hops were reduced, but the effect was more dominant when the pivot factor (threshold value) was small.

3 Concept and System Architecture

This chapter will discuss the concept of our dynamic data migration scheme and will relate it to the state-of-the art discussed in Chapter 2. It will further explain the individual modules used in developing our scheme and will illustrate our system design.

3.1 Concept

In [19], the authors have implemented a dynamic page migration scheme for a multiprocessor architecture which has mesh connection with a distributed-shared global memory as shown in figure 2.2. We differ from their scheme in the sense that they have done data migration at the granularity of pages whereas we want to do it at the granularity of a TLM Block which can vary in size from one cache line to multiple cache lines. Also, our architecture differs from theirs as we have tiles (not processors) connected in a Network on Chip (NoC) as shown in figure 3.1.

In [2], [22] and [15] the authors have done dynamic data placement for the last level cache (LLC) in order to reduce the distance between the data blocks and the core requesting it. We also want to bring data and the core requesting the data close together but we dynamically migrate the distributed shared tile local memory (TLM) instead of the LLC. For that we need to monitor the accesses made to the TLM of every tile in order to decide if migration is needed or not.

3.1.1 Tile Local Memory

The Tile Local Memory (TLM) is a distributed-shared memory in the processor architecture. By distributed-shared we mean that it is distributed among all the tiles and can be accessed by core's placed remotely on neighboring or far away tiles.

Types of Accesses to TLM

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. We have two kinds of accesses to the TLM:

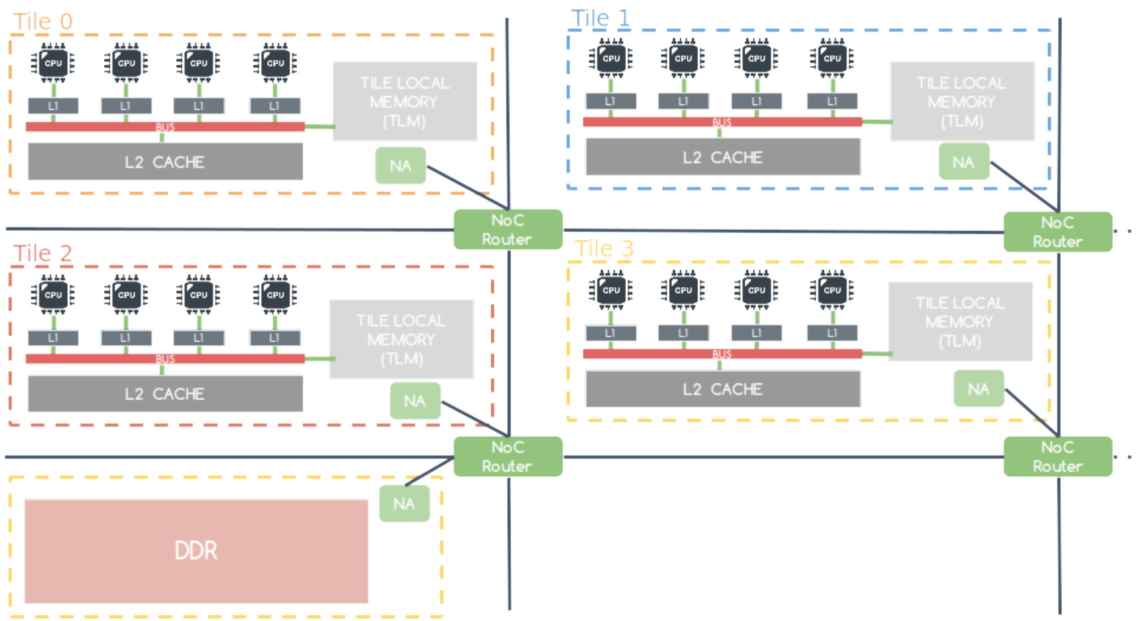

Figure 3.1: Our architecture.

Local TLM Accesses: Figure 3.2 depicts a scenario where a core is accessing its own tile's TLM which makes it a local TLM access. In this scenario the data transfer is happening over the bus inside the tile and there is no traffic going outside the tile through the network adapter. This takes less time as data is placed close to the core that uses it.

Remote TLM Accesses: Figure 3.3 depicts a scenario where a core is accessing another tiles TLM which makes it a remote TLM access. In this scenario the data transfer is happening over the bus, through the network adapter and the NoC Router and global traffic is generated. This takes more time since data is placed far away from the core that uses it.

3.1.2 Goal

We want to reduce the number of remote TLM accesses since it takes more time to fetch data from a TLM which is placed on another tile (as the request has to go on the bus through the network adapter and over the NoC Router in order to go to the other tile) and less time to fetch data which is on a core's own tile's TLM.

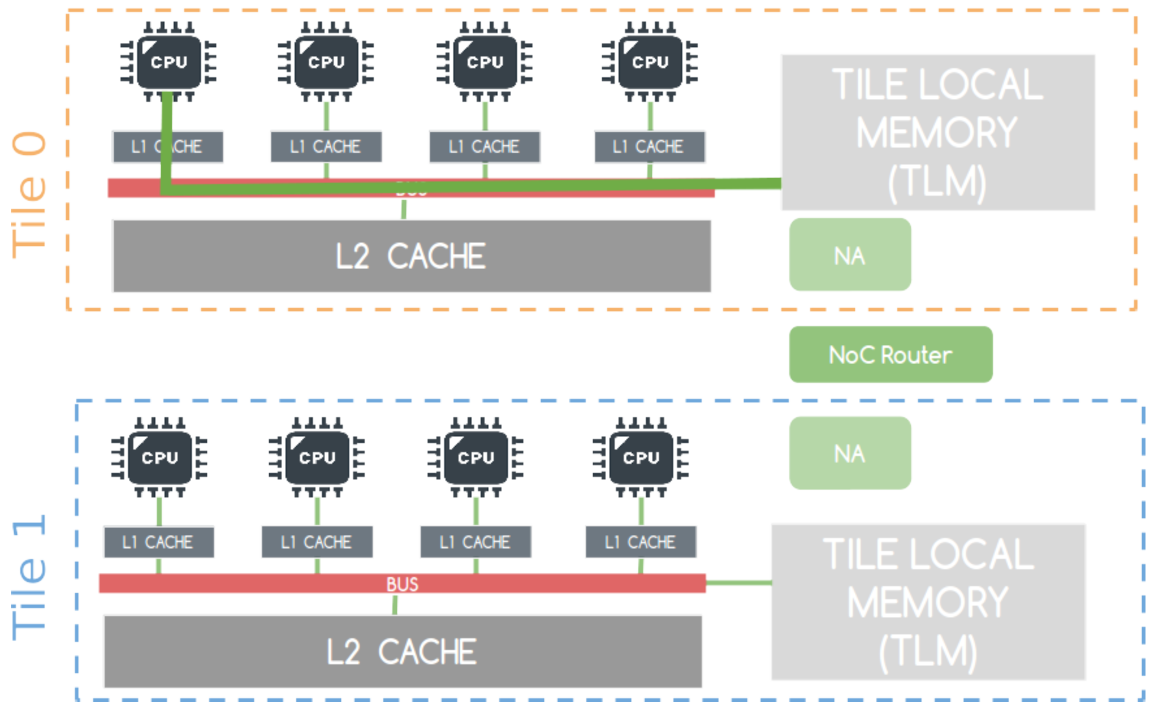

Figure 3.2: A tile depicting local accesses to a TLM.

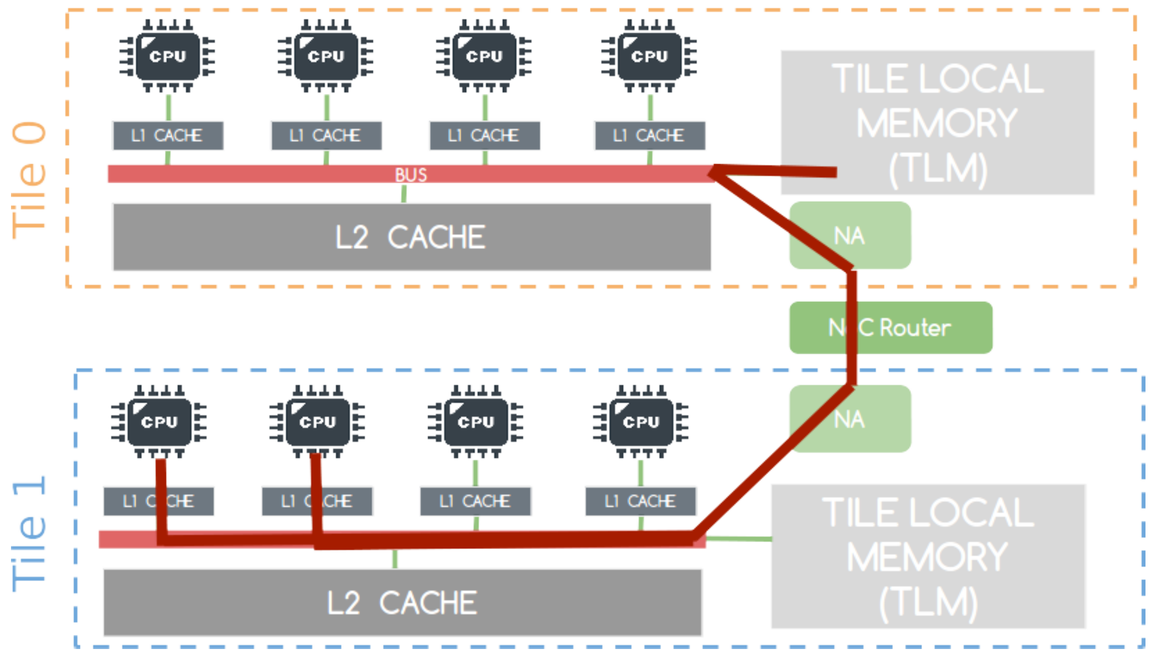

Figure 3.3: A tile depicting remote accesses to a TLM.

3.2 Modules Directly Used in the System for Implementing Thesis

3.2.1 Trace File

As we use a trace based processor model, we first generate and save the traces using the gem5 simulator [16]. We build gem5 for Alpha ISI and use the modified Linux kernel and disk image from [13] to execute benchmarks in full system mode [1]. This is basically the input to the simulator for which we want to find out the execution time.

3.2.2 Memory Management Unit

The Memory Management Unit (MMU) sits between the trace file and the CPU cores as shown in Figure 3.5. This unit is basically a table which contains the address translation of all the addresses from the DDR to the TLM. At start all TLM's are empty which means every instruction has to access data from the DDR. The memory management unit is updated if the DDR is accessed or if a migration triggered by the Central Stats Module takes place.

3.2.3 Cache Stats Module

The Cache Stats module is connected to the L1 and L2 Caches and is continuously getting updates from them regarding cache hits and misses per cache line. This module is responsible for calculating the cache hits and misses per TLM Block. It also calculates the number of local and remote accesses for every TLM block. Cache Stats module also plays a important role in carrying out the data migration command as the migration command has to go through this module.

3.2.4 Tile Local Memory Module

This is the Tile's Local Memory and has been explained in detail in chapter 2. This module has the functionality to observe itself and calculate whether it is empty, is full or has free space. If it has free space then it can find the starting and ending address of all the free spaces.

3.2.5 Central Stats Module

This is the main central module to whom all the Cache Stats modules and TLM Mem modules report to every given time interval ($T_{interval}$). This module is responsible of finding whether migration shall take place or not. It also send out invalidation commands for the data that has to be moved in case if migration is triggered.

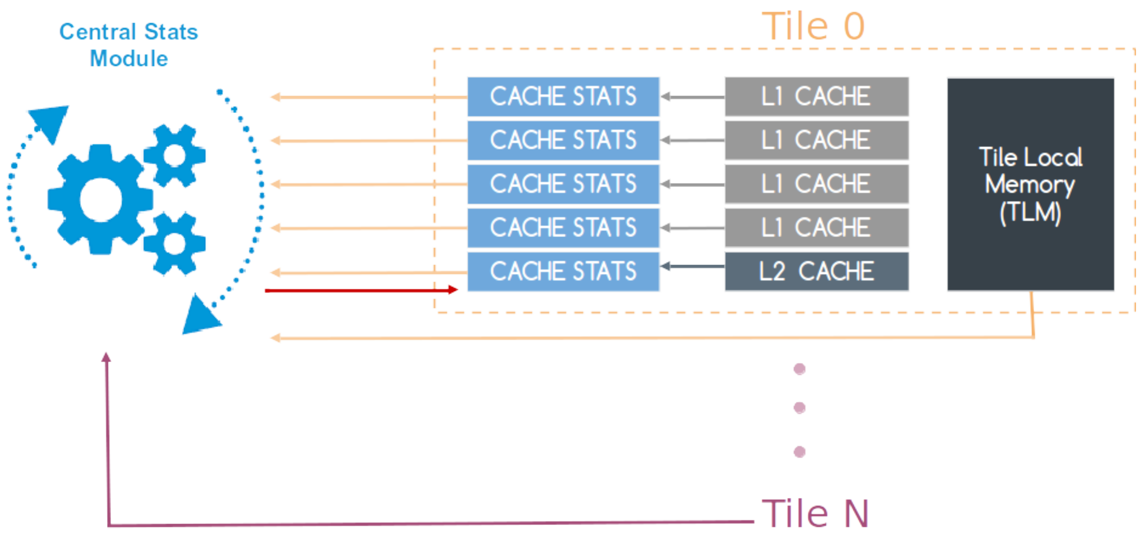

Figure 3.4: Figure illustrating the overview of the modules used.

3.3 System Design

Figure 3.4 shows the overview of the modules involved in dynamic data 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 Stats Module at every given time interval ($T_{interval}$). The Central Stats Module does evaluation of this data and triggers migration if needed.

Also, there is a memory management unit which sits between the trace file and the CPU cores as shown in Figure 3.5. Every trace from the trace file first passes through the memory management unit for address translation and then it is executed.

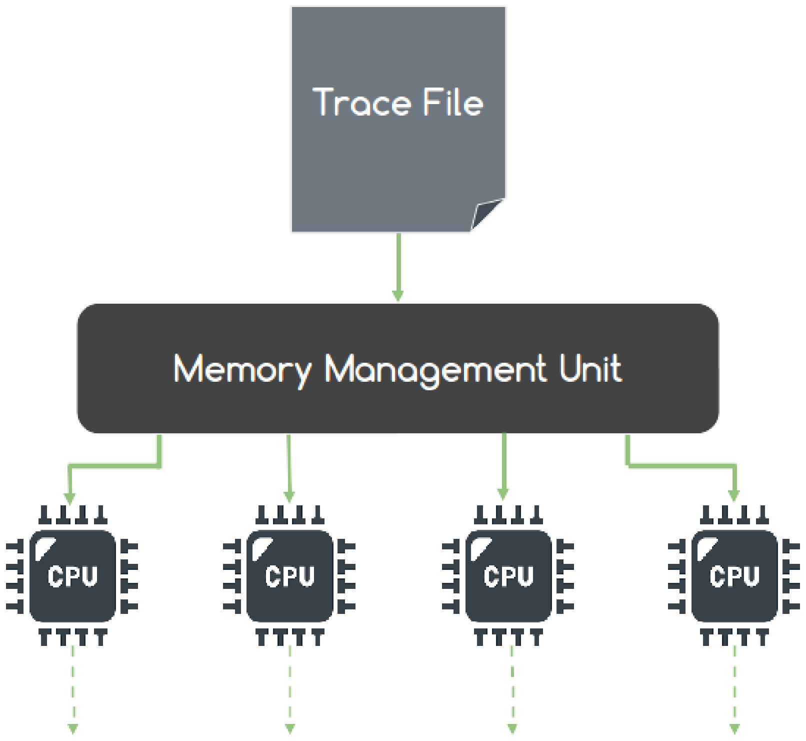

Figure 3.5: Vector Address Table and Trace File Placement

4 Implementation

This chapter will talk about the technologies used to develop the system. It will further discuss the detailed approach of developing the dynamic data migration scheme. We further talk in detail about how the parameters responsible for data migration are calculated and the limitations that exists in the simulator.

4.1 Tools and Technologies

The simulator is created in Synopsys Platform Architect MCO. Synopsys Platform Architect is a SystemC TLM standards-based graphical environment for capturing, configuring, simulating, and analyzing the system-level performance and power of multi-core systems and next generation SoC architectures [21]. The system was programmed in SystemC TLM-2.0 as it is a system-level modeling language. It provides communication-oriented, event-driven simulation interference. TLM is transaction-level modeling in which the details of communication among modules are separated from the details of the implementation of the functional modules [3]. TLM-2.0 focuses on memory mapped bus modeling which is fast and accurate.

4.2 Dynamic Data Migration Process Mechanism

Figure 4.1 shows the messages exchanged between the Cache Stats, TLM Mem and Central Stats Module. At every given time interval ($\mathrm{T}_{interval}$) the Cache Stats module sends the number of local and remote accesses of all the TLM blocks to the Central Stats module and the TLM Mem module sends its free address space to the Central Stats module. The TLM block size is equal to a variable number of cache lines that is determined at compile time by the user. The Central Stats module then evaluates this received data and sends a migrate command to the L2 Cache Stats module if it thinks that migration is needed.

The next subsections will explain how the messages exchanged between the modules are calculated.

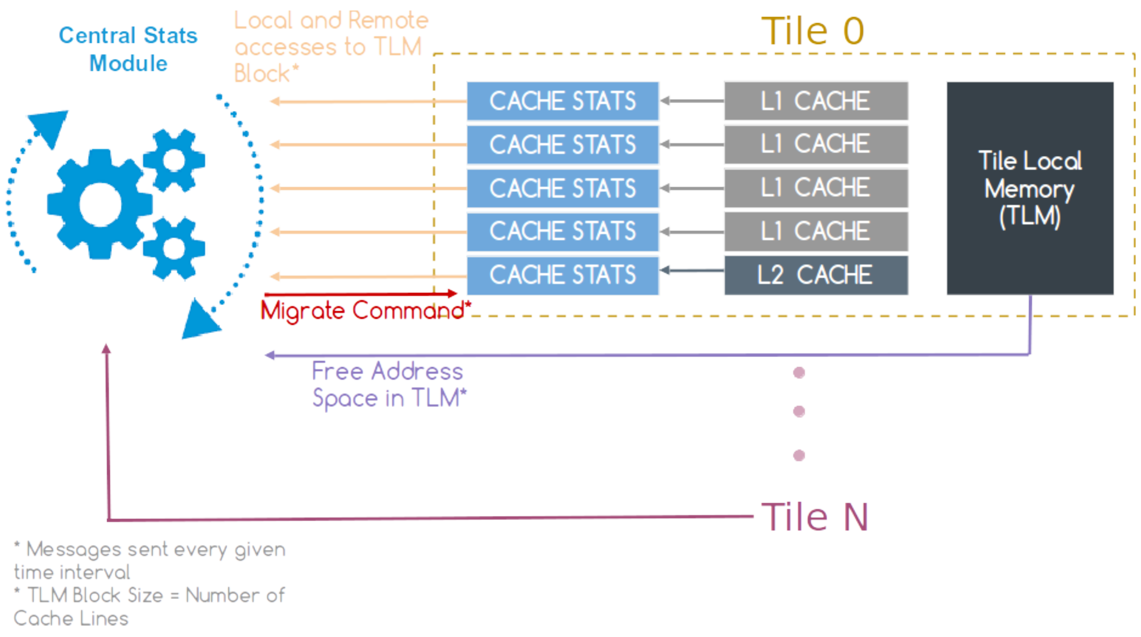

Figure 4.1: Messages exchanged between different modules.

4.2.1 Local and Remote Accesses to a TLM Block

Figure 4.2 shows the calculation behind the metric of local access and remote access to a TLM Block in the Cache Stats module. We compare the TLM number which is being accessed by a instruction with the current tile number (the tile where the cache stats module is placed). If the two values are equal it means it is a request to the TLM of the same tile which signifies it is a local access. If the two values are different it implies it is a request for another tile's TLM and we check whether there is a L1 cache hit or miss. If there is a L1 cache hit, it is a local access. However, if it is a L1 cache miss then we check whether it is a L2 cache hit or miss. If it is a hit then it a local access but if it is a miss then we have a remote access.

4.2.2 Free Address Space in TLM

Figure 4.4 shows how we find the free address space in the TLM MEM module. For calculating the free address space in the TLM we can have three scenarios.

1. TLM Empty
2. TLM Partially Full
3. TLM Full

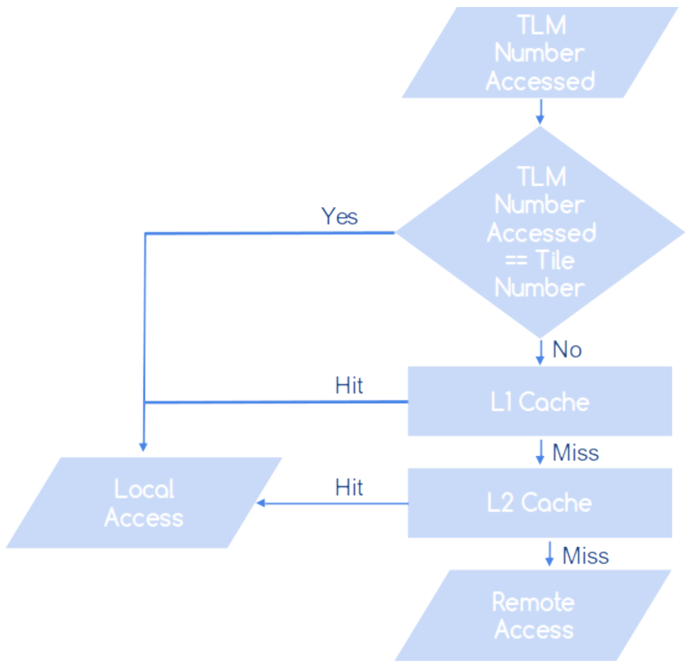

Figure 4.2: Flowchart illustrating how to determine a local and remote TLM access.

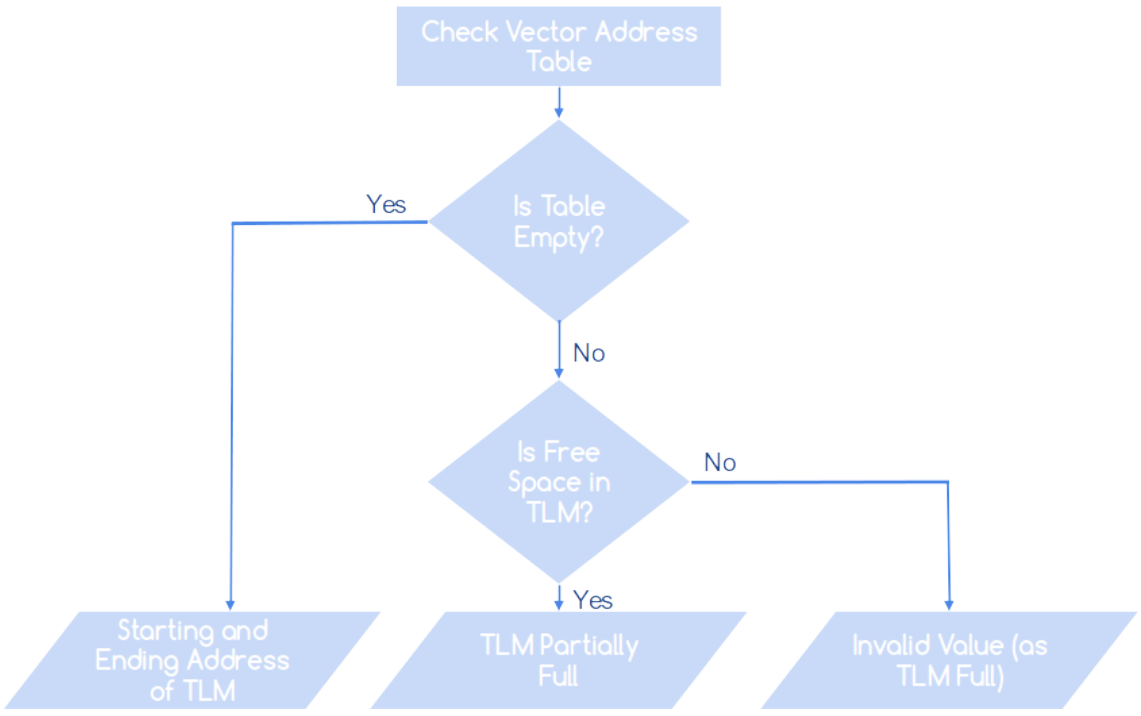
Check Vector Address Table
Is Table Empty?
Yes
No
Is Free Space in TLM?
No
Yes
Starting and Ending Address of TLM
TLM Partially Full
Invalid Value (as TLM Full)

Figure 4.3: Flowchart depicting how the free address space in the TLM is calculated.

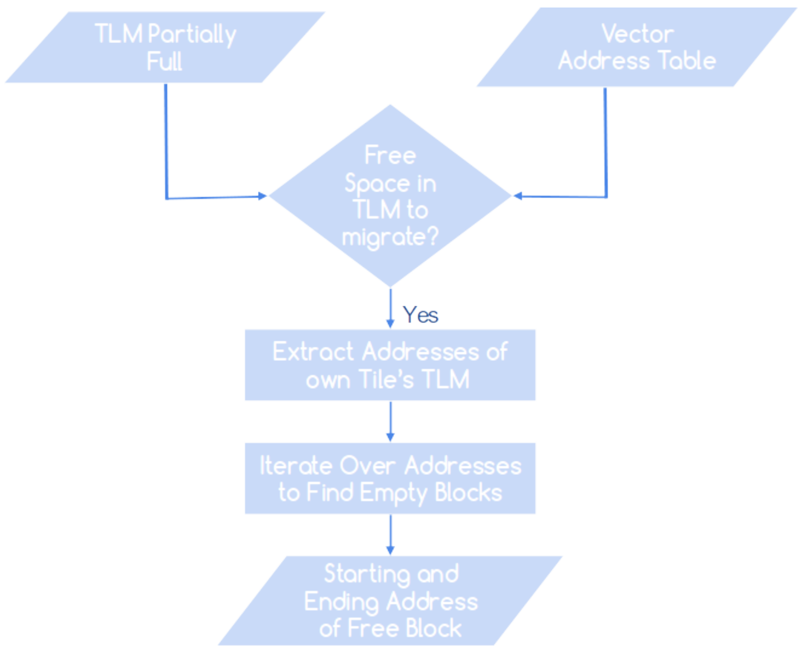
TLM Partially Full
Vector Address Table
Free Space in TLM to migrate?
Yes
Extract Addresses of own Tile's TLM
Iterate Over Addresses to Find Empty Blocks
Starting and Ending Address of Free Block

Figure 4.4: Flowchart showing how the free address space in TLM is calculated if the TLM is partially full.

TLM Empty

If the vector address table in the memory management unit is empty then we know straightaway that the TLM is empty. In that scenario, the starting and ending address of the TLM address space are sent to the Central Stats Module.

TLM Partially Full

If the vector address table in the memory management unit is not empty we iterate over the table and extract the addresses belonging to the current tile's TLM. After we have these addresses we analyze them and determine the free address space in the TLM. The starting and ending address of one free space block is send to the Central Stats Module.

TLM Full

In this case also the vector address table in the memory management unit is not empty. We extract the addresses belonging to the current tile's TLM and if there isno space in the TLM we inform the Central Stats Module this by sending a invalid value.

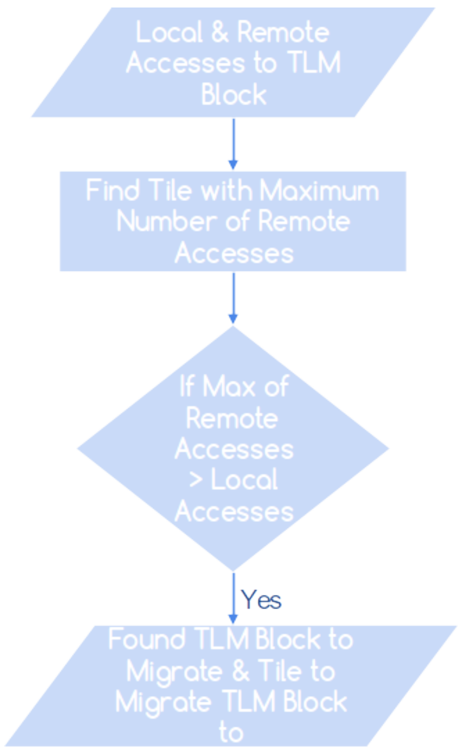
Local & Remote Accesses to TLM Block
Find Tile with Maximum Number of Remote Accesses
If Max of Remote Accesses > Local Accesses
Yes
Found TLM Block to Migrate & Tile to Migrate TLM Block to

Figure 4.5: Process to determine the TLM Block to migrate and the tile to migrate the TLM Block to.

4.2.3 Triggering Migrations

Which TLM Block to Migrate and Tile to Migrate it to?

Figure 4.5 shows the the tile to migrate a certain TLM block is determined. In the Central Stats Module first it is determined whether a TLM Block shall be migrated or not and if it has to be migrated then it is decided which tile to migrate it to. This decision is based on the local and remote accesses to the specific TLM block (explained in detail above). For every TLM Block, the tile with the maximum remote accesses is found and if these remote accesses are greater than local accesses it means it has to be migrated to this tile.

When to Trigger Migrations?

Figure 4.6 shows the algorithm for determining when migration shall take place. After we know the tile to which a specific TLM Block shall me migrated, it is determined whether there is free space in that tile's TLM. If there is free space, a migration command is send to L2 Cache Stats module.

However, if there is no free space in the TLM, then the block with the least number of local accesses in the TLM is identified and this number of local accesses is compared with the number of remote accesses of the TLM block which is to be migrated. If the number of remote accesses of the TLM block to be migrated is higher than this number of local accesses a migration command is send to the L2 Cache Stats module for this block in-order to migrate it to the DDR 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 for the TLM block to be migrated to the tile. If the remote accesses of the TLM block to be migrated are less than the minimum local accesses in the TLM then the migration is flushed.

In case all the local accesses to a TLM are equal and a minimum cannot be found then at random a TLM Block is picked and the local accesses are compared with the migrating TLM Block's remote accesses. If the former is smaller than the latter, the TLM block is migrated 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 for the TLM block to be migrated to the tile.

Migrate Command

The migrate command is split into two commands in the Cache Stats module; first reading data from the location from where data has to be migrated and then writing data at the new location to which the migration is taking place. Once, the data is read a invalidation command is sent to all L1 and L2 caches for that TLM Block and once the data is written the vector address table in the memory management unit is updated.

4.3 Usability Improvement

In the simulator we have now given an option to turn dynamic data migration on and off by setting a parameter from the command line or from within the tool Synopsys. It makes the result gathering phase quicker. Also, since the TLM Block size and the time interval ($T_{interval}$) are variable and can be changed by the user; the change in performance or execution time by varying these two parameters can be analyzed.

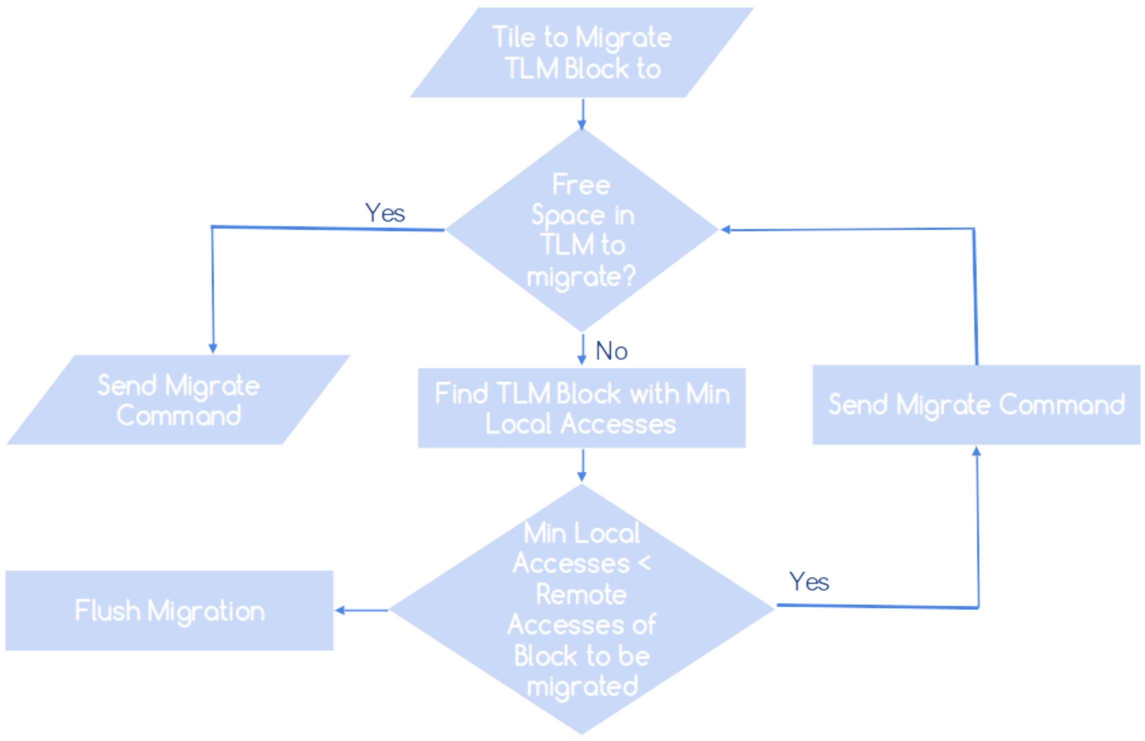
Tile to Migrate TLM Block to
Free Space in TLM to migrate?
Yes
No
Send Migrate Command
Find TLM Block with Min Local Accesses
Send Migrate Command
Min Local Accesses < Remote Accesses of Block to be migrated
Yes
Flush Migration

Figure 4.6: Triggering Migration Commands

4.4 Limitations

Adding the dynamic data migration support to the simulator increased the real time of the simulation by quite an extend.

In the simulator the main modules added because of the dynamic data migration scheme are:

- Twenty Cache Stats modules
- One Central Stats module
- One Vector Address Table

The memory needed for these modules also add quite an overhead to the system.

complexity of the algorithm

Module	Size	Memory Usage	Resource Utilization
Cache Stats Module	xx	xxx	xxxx
Central Stats Module	xx	xxx	xxxx
Memory Management Unit	xx	xxx	xxxx
Total	xx	xxx	xxxx

Table 4.1: Table showing the size of the modules

5 Experimental Setup

This chapter will lay out the programs that assisted in running the simulations. Moreover, it will also illustrate the specifications of the machine that were utilized to run the experiments.

5.1 Size of Modules

Table 5.1 shows the size of the Tile Local Memory, L1 Caches and L2 Caches used in the simulations.

Module	Size	Memory Usage	Resource Utilization
TLM	xx	xxx	xxxx
L1 Cache	xx	xxx	xxxx
L2 Cache	xx	xxx	xxxx
Total	xx	xxx	xxxx

Table 5.1: Table showing Size of TLM, L1 Cache and L2 Cache

5.2 Properties of Modules

5.3 The gem5 Simulator and Trace Files

The gem5 simulator [16] is a modular platform for computer system architecture research, encompassing system-level architecture as well as processor micro-architecture. We use the gem5 full system simulator to generate the trace file for the simulations. For all our simulations, we make use of workloads from the PARSEC Benchmark Suite [1]. We choose four parallel workloads covering three application domains and generate their trace files:

- Blackscholes, Swaptions (Financial Analysis)
- Canneal (Engineering)
- Fluidanimate (Animation)

5.4 Platform Architect MCO

The simulator is designed in Synopsys Platform Architect MCO [21]. It gives a graphical environment for configuring and analyzing the system.

5.5 Writing Shell Script

A shell script for running the migration command is written so that multiple simulations can run in parallel without the need to recompile the simulator. The maximum number of parallel simulations that we could run was four. This saved us considerable time in gathering results.

5.6 Nice Command

The script is run with *nice* command [9] which runs the simulation with a different priority then the usual. We ran the simulations with a increased priority of nine.

5.7 Output Files

The output of the simulation is saved in different text files. The files important for this thesis are:

- Screen Log File
- Time Log File

5.7.1 Screen Log File

The screen log file gives the virtual processing time (in nano seconds) of each processor and also of the entire system. It also gives the real time, the user time and the system time of running the simulations in minutes and seconds. Further it gives the total number of remote reads and remote writes of the simulation run and some other parameters and queue sizes that are not relevant for this thesis.

5.7.2 Time Log File

The time log file consists of the breakdown of the virtual time (in nano seconds) that each processor took executing instructions on different tiles and on the individual components/modules like TLM, L1-Cache, NPC, NoC, Bus etc.

5.8 System Specifications

The simulations were run on a core i5 Intel processor with a Solid-State-Drive (SSD) —? in-order to reduce the real time for the simulations. Ubuntu..

6 Evaluation

This chapter illustrates the results gathered from the simulator. It gives the comparison of the execution time of the simulator with data migration on against the first touch and most accessed data placement. It also shows the effect on execution time of different TLM block size.

6.1 Dynamic Data Migration vs First Touch and Most Accessed

In this section the outcome of the simulation is analyzed and the performance of every workload is compared. Basically the execution time when dynamic data migration is turned on is compared with the execution time of data placed via first touch and most accessed schemes which have been explained in section xxx for every workload.

6.1.1 Blackscholes

6.1.2 Swaptions

6.1.3 Fluidanimate

6.1.4 Canneal

6.2 Dynamic Data Placement for Different TLM Block Size

In this section the output of the simulator is analyzed for different TLM Block sizes.

6.3 Dynamic Data Placement for Different Time Interval ($T_{interval}$)

In this section the output of the simulator is analyzed for different Time Interval ($T_{interval}$).

7 Conclusion and Outlook

This chapter will summarize the work done in the thesis and will give insights into how it can be used in future research.

7.1 Summary

In this thesis a dynamic data migration scheme has been introduced for a distributed-shared memory architecture with multiple cores and multiple tiles. It describes how the data can be migrated from one tile's local memory to another tiles local memory at run time. As a result, we don't need to find the best placement for the memory before running the application.

7.2 Future Work

Bibliography

[1] Thomas Wild Akshay Srivatsa, Sven Rheindt and Andreas Herkersdorf. Region based cache coherence for tiled mpsocs. System-on-Chip Conference (SOCC), 2017 30th IEEE International, Sept. 2017.

[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] L. Cai and D. Gajski. Transaction level modeling: An overview. page 19–24. proceedings of the Int. Conference on HW/SW Codesign and System Synthesis (CODES-ISSS), Oct. 2003.

[4] D. W. Clark and J.S Esner. Performance of the vax-11/780 translation buffer: Simulation and measurements. volume 3, pages 31–62. ACM Trans. Comput. Syst, Feb. 1985.

[5] 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.

[6] DFG. Welcome to the pages of the tcrc 89.

[7] 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.

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

[9] IBM. nice - run a command at a different priority. `https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.3.0/com.ibm.zos.v2r3.bpxa500/nice.htm`.

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

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

[12] 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.

[13] E. Fatehi P. Gratz M. Gebhart, J. Hestness and S. W. Keckler. Running parsec 2.1 on m5, October 2009.

[14] 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.

[15] 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.

[16] Gabriel Black Steven K. Reinhardt Ali Saidi Arkaprava Basu Joel Hestness Derek R. Hower Tushar Krishna Somayeh Sardashti Rathijit Sen Korey Sewell Muhammad Shoaib Nilay Vaish Mark D. Hill Nathan Binkert, Bradford Beckmann and David A. Wood. The gem5 simulator. volume 39 Issue 2, pages 1–7. ACM SIGARCH Computer Architecture News, May 2011.

[17] 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.

[18] Sven Rheindt. inetworkadapter documenation, 2016.

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

[20] Christoph Scheurich and Michel Dubois. Dynamic memory allocation in a mesh-connected multiprocessor. pages 302–312. Proc. the 20th Hawaii Int. Conf. Syst. Sci., Jan. 1987.

[21] Synopsys. Soc architecture analysis and optimization for performance and power. `https://www.synopsys.com/verification/virtual-prototyping/platform-architect.html`.

[22] 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 21, 2018

. . . .
Iffat Brekhna