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

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.

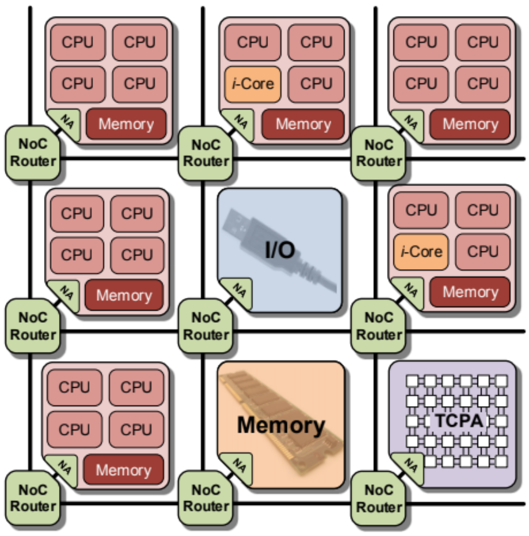

Figure 1.1: Invasic Architecture [15]

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 a large 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. Another drawback is that the real time to run the simulation might increase as a large number of computations need to be done to run the dynamic data placement algorithm.

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 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.

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 [8], [15]. 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.

Bus

The bus connects the cores to the L2 Cache and the TLM and to other tiles. When accessing another tile the request has to go through a network adapter to be routed to the destination tile. We are using a AMBA Bus ....Iffat

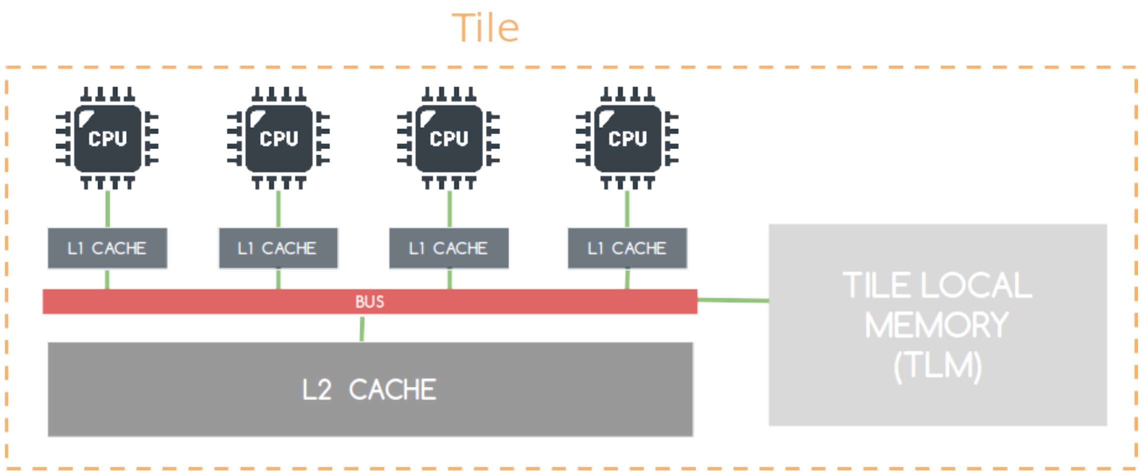

Figure 2.1: A Tile

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 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 [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], [6] 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], [18], [12] 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 [12], [14]. 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 [14].

There is a hybrid data placement [14] 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 [14]. 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 [5]. However, finding the optimal mapping between threads and cores is a NP-hard problem [9] 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 [10]. In [10] 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 [4] 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 [4].

2.2.3 Data and Thread Migration

In [11] 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 [11].

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 Placement and Migration on TLM

In [16] the authors have proposed a dynamic page migration scheme for a multi-processor 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

The authors collected their results by seeing how the dynamic page migration scheme improved their number of hops. In all the cases the hops were reduced, but the effect is more dominant when the pivot factor (threhold value) is small.

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

3 System Architecture

This chapter will illustrate and explain the architecture and design of the system. It will further explain each module in detail.

3.1 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.

3.1.1 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.

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. In this scenario the data transfer is happening just inside the tile and there is no traffic going outside the tile to the bus. This takes less time as data is placed close to the core that uses it.

Remote TLM Accesses

Figure 3.2 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 and traffic is generated. Also, 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 places on another tile (as the request has to go on the bus through the network adapter to the other tile's bus) and less time to fetch data which is on a core's own tile.

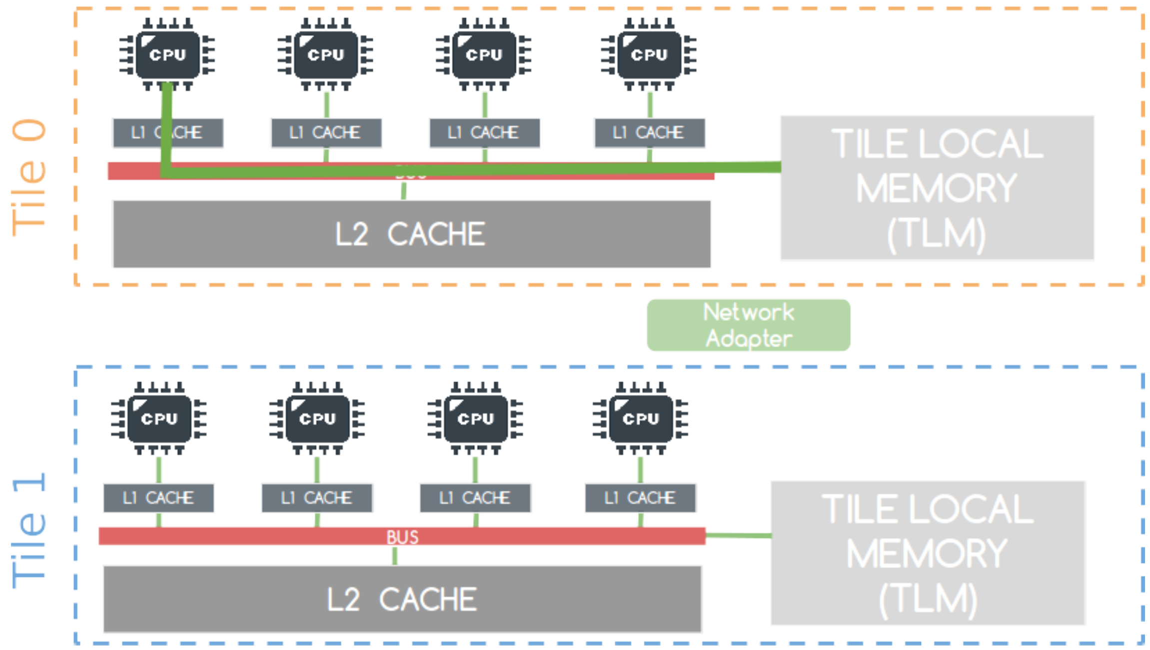

Figure 3.1: A tile showing local accesses to a TLM

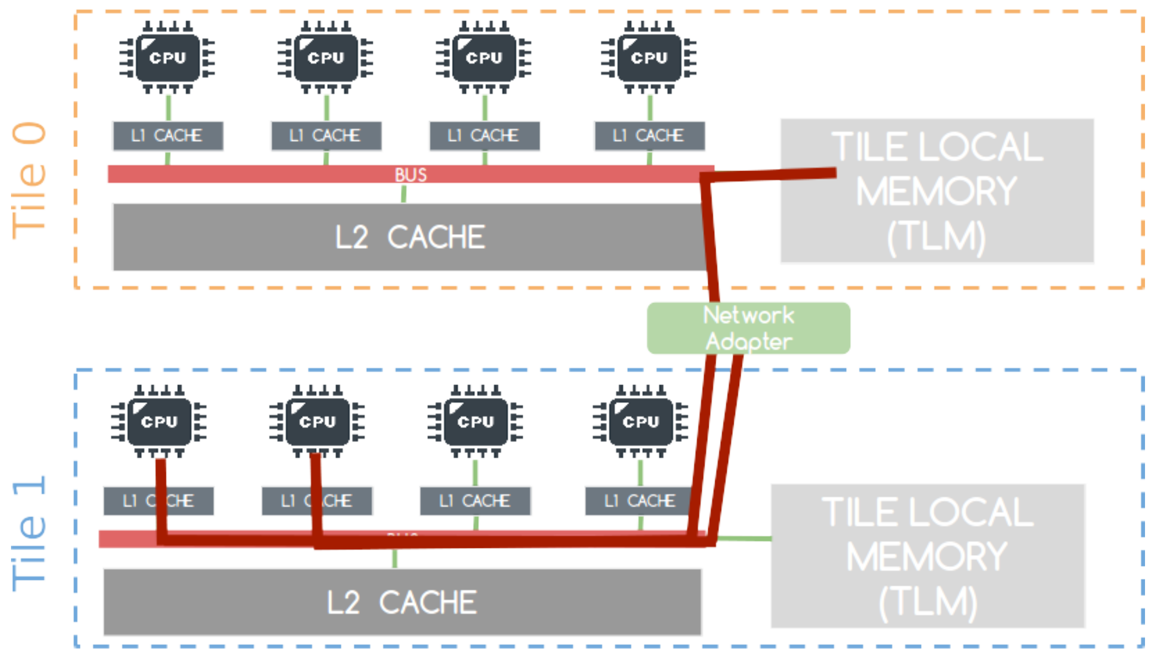

Figure 3.2: A tile showing remote accesses to a TLM

3.2 Modules Directly Used in the System for Implementing Thesis

3.2.1 Trace File

How to explain ask from Sven!
We run a benchmark in gem5 full system simulator [13] to generate the trace-file which —.

3.2.2 Vector Address Table

The vector address table sits between the trace file and the core's as shown in Figure 3.3. 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 Stats Module and if that migration 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 and to calculate for every TLM block the number of local and remote accesses. It 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 TLM Mem Module

This is the Tile's Local Memory and has been explained in detail in chapter 2. This module also 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 staring and ending address of all the free spaces.

3.2.5 Central Stats Module

This is the main central module which all the Cache Stats modules and TLM Mem modules are reporting to. This module is responsible of finding whether migration shall take place or not and to send out invalidation commands for the data that has to be moved in case if migration has to be triggered.

3.3 System Design

Figure 3.4 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 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 vector address table which sits between the trace file and the CPU's as shown in Figure 3.3. Every instruction from the trace file first passes through the vector address table for address translation and then it is executed.

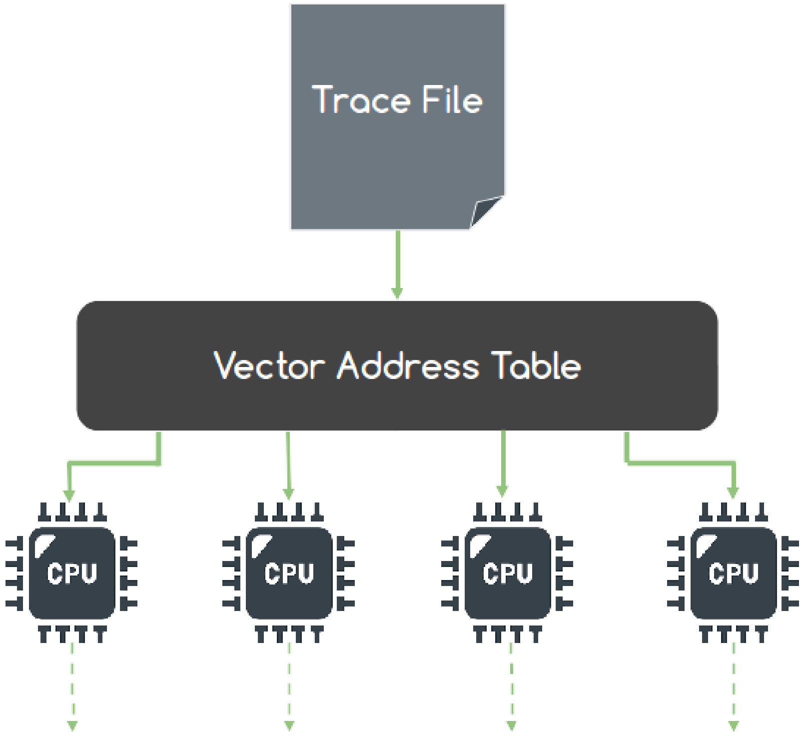

Figure 3.3: Diagram showing where the trace file is placed

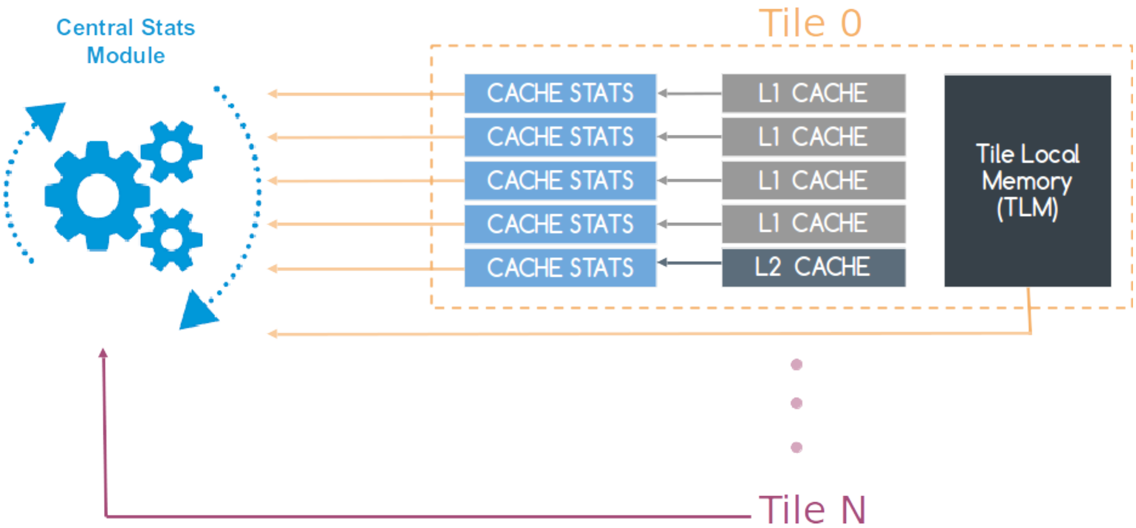

Figure 3.4: Diagram Showing Overview of the Modules Used

4 Implementation

In this chapter, the technologies, detailed approach of the system and the design used to develop the simulator have been discussed. We further talk in detail about what 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 [17]. 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 Concept

Figure 4.1 shows the messages exchanged between the Cache_Stats, TLM Mem and Central Stats Module. At every given time interval ($T_{interval}$) 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.

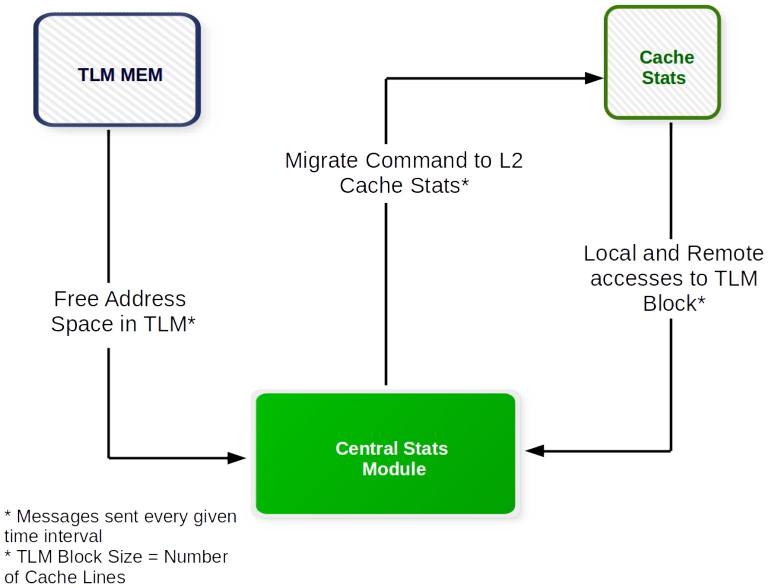

Figure 4.1: Diagram illustrating the messages exchanged between different modules

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

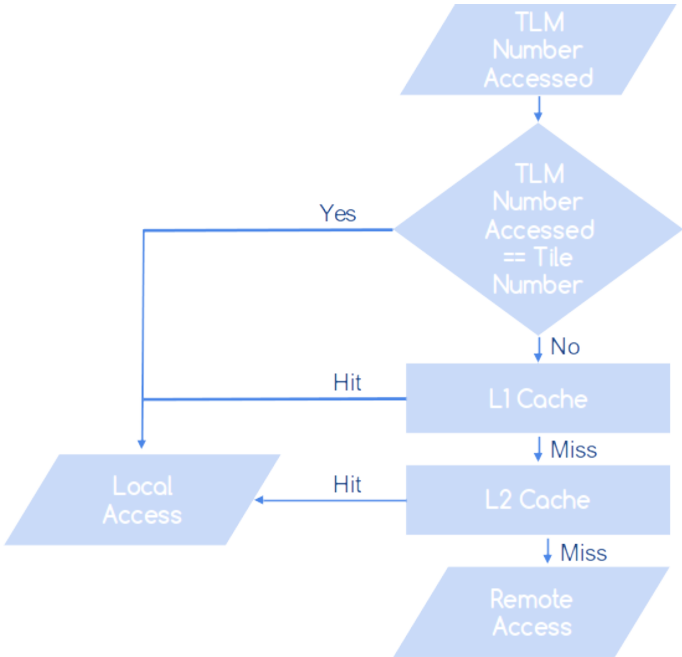

Figure 4.2: Flowchart illustrating how to determine a local and remote TLM access

TLM Empty

If the vector address table 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 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 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.

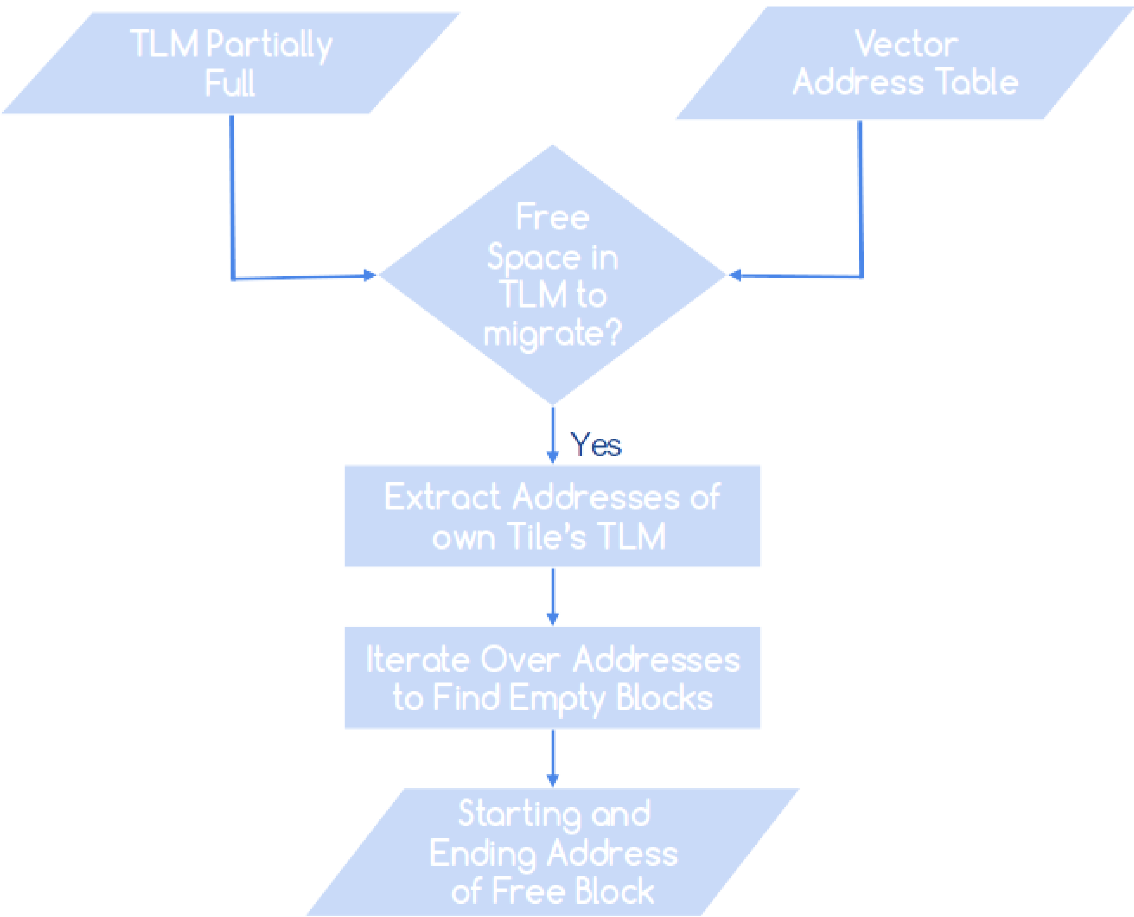

Figure 4.3: Flowchart showing how the free address space in TLM is calculated if the TLM is partially full

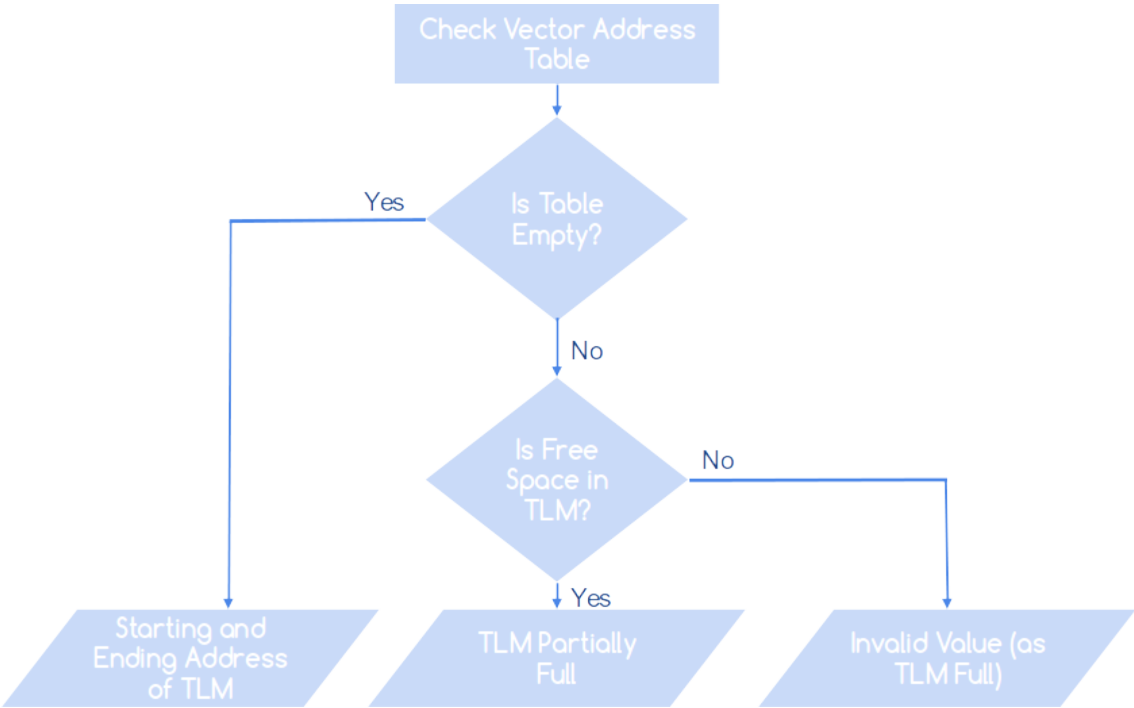

Figure 4.4: Flowchart showing how the free address space in TLM is calculated

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 com-

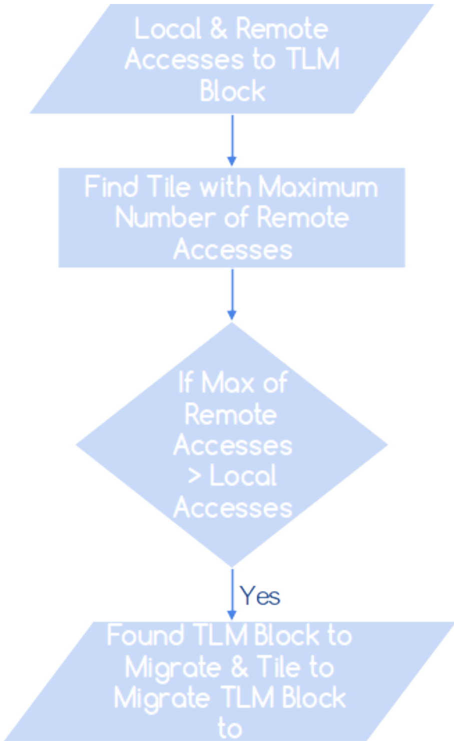

Figure 4.5: Process to determine the TLM Block to migrate and the tile to migrate the TLM Block to

pared 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 is updated.

4.3 Limitations

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

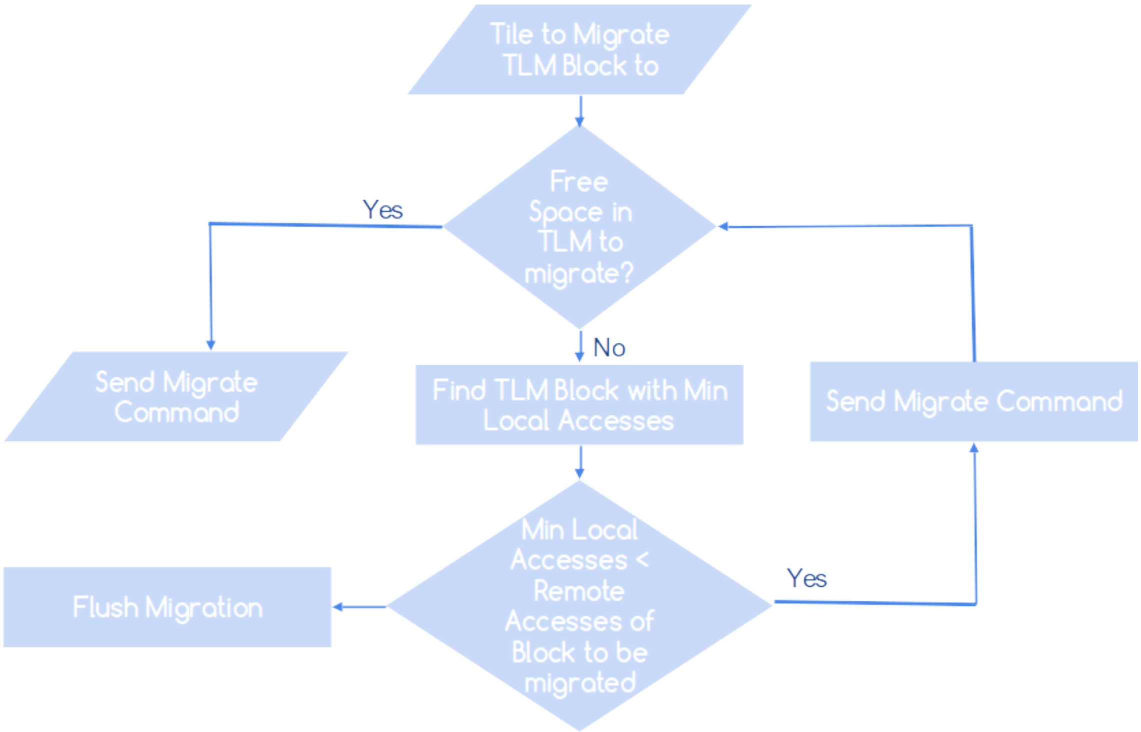

Figure 4.6: Triggering Migration Commands

5 Experimental Setup

This chapter will lay out the programs that assisted in running the experiments on the Synopsys Simulator. ??Moreover, it will also illustrate the specifications of the machine that was utilized to run the experiments.

5.1 The gem5 Simulator

The gem5 simulator [13] 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.

5.2 Platform Architect MCO

The simulator is designed in Synopsys Platform Architect MCO [17]. It gives a graphical environment for configuring and analyzing the system.

5.3 Writing Shell Script

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

5.4 Nice Command

The script is run with *nice* command [7] which runs the simulation with a different priority then the usual. The simulation is run with a increased priority of nine.

5.5 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

5.5.1 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.

5.5.2 Time Log File

The time log file consists of the breakdown of the time (in nano seconds) that each processor took executing instructions on different tiles and on the TLM, L1-Cache, NPC, NoC, Bus etc. In other words it breaks down the time of

5.6 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.

6 Evaluation

7 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] 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] 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.

[5] 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.

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

[7] 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`.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Sven Rheindt. inetworkadapter documenation, 2016.

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

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

[18] 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