NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

A STUDY ON PERFORMANCE HURDLES FOR HETEROGENEOUS 3D-MPSoCs

by

DANG HOANG TUNG
G1601337K

A thesis presented for the partial fulfilment of
the requirements for the Degree of
Master of Science in Embedded Systems

Supervised by
Asst. Prof. Arvind Easwaran

July 2018

Contents

Abstract

Three-Dimensional Multi-Processor System-on-Chip (3D-MPSoC) is gaining popularity as a potential candidate to replace conventional multicore processors in high performance, time and space critical embedded systems. As it stacks up layers of processing cores on top of each other, area occupancy reduces significantly. Moreover, Through-Silicon-Vias (TSVs), which is significantly faster than the communication channel between horizontal cores, can be used as the communication channel between the vertical cores. As a result, parallelization and multiprocessing capability increase enormously.

Despite having multiple benefits, 3D stacking of processing cores introduces several problems that were not present in the conventional multi-core processors. One such problem is Thermal Adversity. 3D-MPSoCs usually deploy only one heat-sink below the processing core stack. Therefore, during execution of an application, temperatures in different layers of the stack vary significantly. To prevent wear out, several techniques such as dynamic voltage and frequency scaling (DVFS), thermal-aware task scheduling, etc. are used. But these techniques (DVFS) vary the processing frequency in different levels of the stack. Furthermore, thermal-aware task scheduling migrates tasks among cores to cool down the temperature of a 3D-MPSoC during runtime. When a task is migrated to a new core, it produces a lot of L1 cache-misses because the processing data is not available in the new core. The L1 cache-misses would trigger requests to L2 cache or higher layer in the memory hierarchy to load the requested data. All of these techniques assume that the conventional processor architecture and its memory subsystem are suitable to adapt them and no change is required in the processor architecture.

In our research, we are interested in verifying the aforementioned assumption. To be specific, we would like to examine whether conventional processor cache memory hierarchies could cause any performance hurdles to data sharing in multi-threaded applications in 3D-MPSoCs. We consider conventional heterogeneous and homogeneous organization of processor cores as a layer of the 3D stack, and its cache memory hierarchy may not suit/benefit the requirements/features of 3D-MPSoC. To the best of our knowledge, no prior work analysed the hurdles imposed by the processor cache memory in 3D-MPSoCs. Our experimental works show that heterogeneous architecture and exclusive cache hierarchy cause substantial performance hurdles to data sharing in multi-threaded applications in 3D-MPSoCs.

Acknowledgment

I am grateful to my supervisor, Asst. Prof. Arvind Easwaran for extremely supporting my project choice and for giving invaluable advice during the time of this dissertation.
I am thankful to Dr. Mohammad Shihabul Haque, for the patience, advice, and support throughout the project. His valued advice has helped me to overcome hurdles encountered along the way.
I would also thank Saravanan Ramanathan for the continuous support and valuable contributions to the project.
Finally, I would like to thank my family for their everlasting support.

List of Figures

Listings

Chapter 1

Introduction

Single core processor technology accelerates processing speed by increasing clock frequency. However, increasing frequency raises individual processors' temperature and results in higher power consumption. This problem is known as clock speed saturation and power wall issue in single-core processor design. The multi-core processor technology emerged as a solution to resolve this issue. Multi-core processor architectures reduce clock frequency in individual cores and improve throughput by utilizing parallel processing. At the same clock frequency, a multi-core processor does process more data than a single core processor.

Due to the advantages of multi-core, the number of cores continues to grow giving rise to new issues. For example, the communication delay between cores which are far from each other is a critical problem. The limitation in the design area of a chip is also necessary to be considered when designers increase the number of processor cores. Emerging technologies have been proposed to address the challenges in multi-core processors such as transactional memory, novel interconnects, and three-dimensional multi-processor System-on-Chip (3D-MPSoC).

3D-MPSoC has gained significant attention recently due to its advantages in various aspects. It stacks two or more dies vertically to increase the device density and reduces the communication latency between dies. Especially 3D-MPSoC technology helps to reduce the interconnection between vertical processor cores significantly by using Through-Silicon-Vias (TSVs). TSVs are vertical electrical connections that pass completely through a silicon wafer or die. It increases the inter-core communication speed and utilizes the processor space efficiently. A designer can place more processor cores in a small area, thus increasing the parallelism. The power consumption in each core is reduced, but the desired performance is still achieved. Figure 1.1 shows an abstract architecture of 3D-MPSoCs.

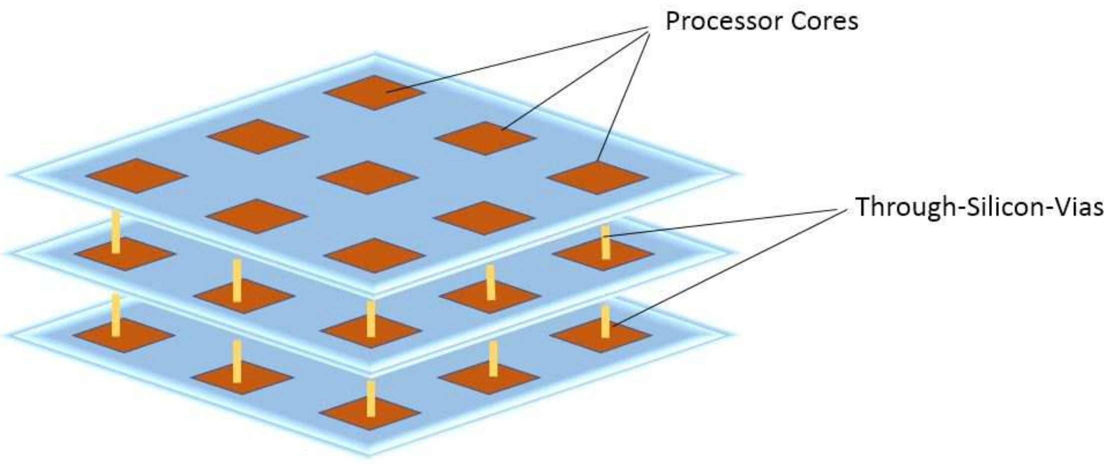

Figure 1.1: 3D-MPSoCs architecture.

One major drawback of 3D-MPSoCs is the excessive heat generation from the internal layers. During overheating, the power density per unit volume increases drastically. When inner cores are active frequently, they create critical hotspots within the chip. These hotspots raise the concern of heat dissipation from processing cores that are far away from the heat sink. Because they do not directly connect to the heat sink, which is attached at the bottom of the chip. High temperatures in a processing core will trigger voltage and frequency throttling in hardware and degrade the processor's performance. Moreover, it reduces the processor's reliability and its lifespan.

Due to thermal adversity, thermal management has become an important topic in 3D-MPSoCs design. There are several research topics that try to solve the thermal issue in 3D-MPSoCs, such as thermal-aware task scheduling in 3D-MPSoCs and Dynamic Voltage and Frequency Scaling (DVFS). DVFS and thermal-aware task scheduling approaches assume that conventional processor architectures can adapt these techniques without changes. But, those solutions do not address the thermal issues completely because DVFS and thermal-aware task scheduling involve task migration which may lead to large overheads. These overheads could contribute a significant amount of energy consumption to the processor. Moreover, general 3D-MPSoCs assume that cores can share data easily (DVFS and task scheduling approaches think the same). Our research works proved that conventional processor cache hierarchy, especially exclusive cache hierarchy does not help in data sharing among processor cores.

In this project, we are interested in finding the suitability of conventional multicore cache hierarchy architectures for 3D-MPSoCs and its essential management techniques such as DFVS and thermal-aware scheduling. If data sharing in the shared cache memory is efficient, the overhead of task migration introduced by DFVS and thermal-aware scheduling would be reduced significantly because a task is migrated to other processor cores but still able to process the required data in a shared cache memory efficiently. To be specific, we analyse the effectiveness/hurdles of data sharing in the shared cache memory among processor cores to understand how it could help inter-thread data sharing in 3D-MPSoCs. We consider existing multicore systems realized using heterogeneous or homogeneous processors. Their memory subsystems utilize various hierarchical orders such as inclusive, exclusive or non-strictly inclusive caches. To the best of our knowledge, no prior work analysed the efficiency of data sharing in conventional multicore cache hierarchies specific to 3D-MPSoCs.

1.1 Goal

Our goal is to study the performance hurdles caused by multicore architectures in 3D-MPSoCs. In particular, we want to study how multi-threading is benefited or hardened by exclusive and inclusive cache hierarchies. Furthermore, we would like to explore how data sharing in a shared cache memory and cache prefetching are benefited by inclusive or exclusive cache hierarchies for multi-threaded tasks in 3D-MPSoCs.

1.2 Organization of the Dissertation

The remaining chapters of the dissertation will be organized as follows:

- **Chapter 2** introduces cache hierarchy orders in existing multi-core architectures and how it could impact the performance of multi-core systems and 3D-MPSoCs.

- **Chapter 3** presents a literature survey and related works to overcome the thermal issues in 3D-MPSoCs.
- **Chapter 4** gives an overview of Odroid-XU4 and Intel NUC architectures which are the two typical architectures for heterogeneous and homogeneous multicore systems. It also provides the experimental setup on these platforms.
- **Chapter 5** presents the experimental method and explains the wrapper code which is used for the experiments.
- **Chapter 6** shows our experimental results on Odroid-XU4 and Intel NUC.
- **Chapter 7** summarizes this thesis towards 3D-MPSoCs and discusses possible future works.

Chapter 2

Background

This chapter presents some background of cache architectures in multicore systems and explains why cache memory is important to processor's performance as well as its energy efficiency. A key determinant of overall system performance and energy efficiency is the cache hierarchy since access to off-chip memory (external memory) consumes many more cycles and energy than on-chip memory accesses. In addition, multicore processors are expected to place ever higher bandwidth demands on the memory system. All these issues make it important to avoid off-chip memory access by improving the efficiency of the on-chip cache memory. This chapter inspires our research topic by raising the question of how to select the suitable cache memory hierarchy and its policy to avoid memory operations becoming the performance bottleneck in 3D-MPSoCs.

2.1 Cache Architecture of Multicore Processors

Cache memories are small, high-speed buffer memories, which contain the most recently used portions of main memory to fill in the gap between processor speed and memory speed. (Memory speed is much slower compared to processor speed). Most modern multicore processors incorporate multiple levels of the cache hierarchy. Figure 2.1 shows an example organization of multicore cache architecture.

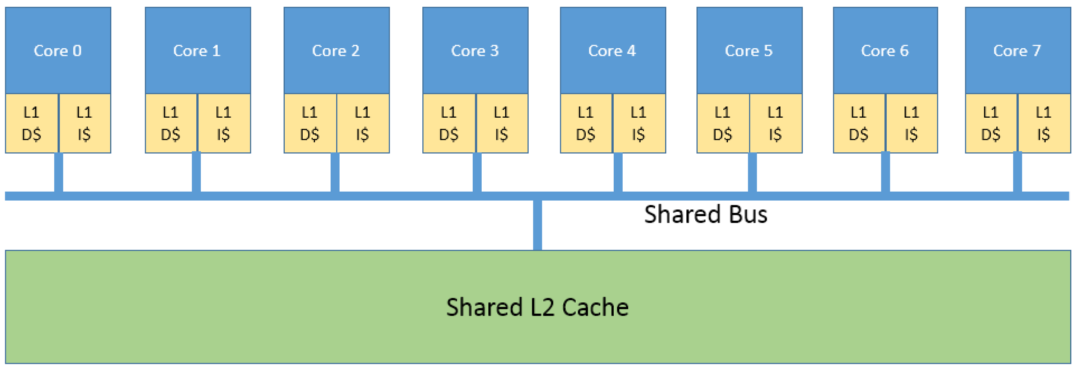

Figure 2.1: Multicore cache organization with a large shared L2 cache and private L1 caches per core.

In a multicore processor, each core typically has its own private L1 data cache and L1 instruction cache. A processor core must access its data/ instruction in the L1 cache first. If there is a cache-miss in the L1 cache, it will initiate a request to the L2 cache. For most of the discussion, we will assume that L2 cache is the LLC (Last Level Cache). However, there is a possibility that L3 is the LLC in a three-level cache hierarchy.

Multi-level cache architectures can be designed in various hierarchies such as inclusive cache, exclusive cache or non-inclusive cache. If the L1-L2 hierarchy is inclusive, it means that every block in L1 has a backup copy in L2. The inclusive policy ensures that when a block is evicted from L2, its copy in the L1 cache is also evicted. If a single L2 cache is shared by multiple L1 caches, the copies of all L1s are also presented in L2. The primary advantage of an inclusive cache hierarchy in a multicore system is the ease in locating a data block upon an L1 cache-miss. Either a copy of the block will be found in L2, or the L2 will point to a modified version of the block in some L1, or the L2 miss will indicate that the request can be sent directly to the next level of the memory hierarchy. The disadvantage of an inclusive hierarchy is the wasted space because most L1 blocks have redundant copies in L2.

On the other hand, some L1-L2 hierarchies are designed to be exclusive (a data block will be found in either an L1 cache or the L2 cache, but not in both) or non-inclusive (there is no guarantee that an L1 block has a backup copy in L2). Data block search is more complex in this setting. On an L1 miss, other L1 caches and L2 will have to be looked up. To implement this look-up method, a cache snooping-based protocol is deployed between L1 and L2. However, with snooping-based protocols, the search operation becomes complicated and does not scale well as the number of L1 caches is increased. Snooping-based protocols also introduce more processor overhead due to the search operations in entire cache space. The advantage is the higher overall cache capacity because there is little (or no) duplication of blocks.

2.2 Cache Performance Analysis

Rapid development in multicore processor improved the processing speed by increasing the number of processor cores. Multicore processors distribute the computation across several processor cores, and high performance could be expected. However, this high performance is based on an assumption that the main memory does not stall the processor cores for acquiring the requested data. In a real world, nevertheless, this cannot be guaranteed. First, in multicore architectures, the L1 cache is smaller than L2 cache. Second, several cores share the same L2 cache memory and as a consequence, an access to the L2 cache memory can take longer time due to the potential bus contention. These facts indicate that on multicore processors more cache-misses can be produced and an off-chip memory reference is more expensive in term of energy consumption. In another word, a cache hierarchy with effective data sharing in an L2 cache will result in better performance and energy efficiency.

Several research studies [1–3] have shown that among all the chip components, the on-chip memory subsystem (caches) consumes the major portion of the whole power consumption. Furthermore, caches in multicore systems are organized into multiple levels, and the Last Level Cache (LLC) consumes a principal amount of chip power [1,4]. Therefore, understanding the advantages and drawbacks in existing multi-level cache hierarchies would help to understand their suitability for use in 3D-MPSoCs.

Chapter 3

Literature Review

In this chapter, we present a survey of research articles which proposed techniques to prevent performance degradation in 3D-MPSoCs due to thermal issues. The common techniques are Dynamic Voltage and Frequency Scaling (DVFS) and thermal-aware task scheduling in 3D-MPSoCs. Thermal-aware scheduling is usually combined with DVFS to minimize the peak temperature in 3D-MPSoCs. In the following sections, we discuss these topics in detail.

3.1 Thermal-Aware Task Scheduling in Heterogeneous 3D-MPSoCs

3D-MPSoCs offer great performance and flexibility. However, due to the increased power density, thermal challenges in 3D-MPSoCs are critical. The thermal problem, i.e., the high temperature will increase leakage power and decrease system performance, reliability and lifetime. Therefore, several research studies have been proposed with the solution of thermal-aware task scheduling in 3D-MPSoCs [5–11]. Thermal-aware task scheduling combines intelligent task mapping with thermal characteristics of 3D architecture to reduce critical hot-spots. Thermal-aware scheduling is usually combined with DVFS to reduce the peak temperature of the system efficiently. DVFS technique is the adjustment of the voltage level and operating frequency of processor cores to optimize resource allotment for tasks and minimize power consumption.

Xiuyi Zhou et al. [5] proposed a scheduling algorithm based on the baseline Linux 2.6 scheduler. Observation shows that the baseline Linux scheduler uses OS scheduling at 100ms interval, which may lead to extremely high temperature when two or more compute-intensive tasks are allocated on the same core stack, and they execute in the stack for a long period (100ms) until the next scheduler invocation. The paper proposed an improved version of Round-Robin scheduler. A Round-Robin scheduler can overcome the issue by rotating tasks among cores with a fixed interval of 8ms to avoid the uneven distribution of power and temperature. The paper also suggested a technique for temperature balancing. The technique groups each core in the same vertical line into one stack (i.e., super core). It sorts the power consumption of all super-tasks (i.e., tasks in the same vertical cores) from low to high and sorts the temperature of all stacks from high to low. The technique does mapping between super-tasks and super cores. In each super core, the task with the highest power consumption will be allocated to the core near the heat sink. When a super core is overheated, the thermal management will select the core with the highest power consumption to engage DVFS. However, the paper did not take into account the overhead of switching algorithm when it runs with a high switching frequency. The other drawback is the grouping of tasks into super cores did not consider the relationships between tasks. Tasks that need to communicate frequently would need to be allocated to cores near to each other.

In another proposal, V.Chaturvedi et al. [7] presented a two-stage thermal-aware mapping algorithm which takes into consideration the periodic constraints of an application, thermal issues, task characteristics and leakage power. This paper introduces an off-line/online thermal-aware task mapping algorithm for peak temperature minimization. It is a two-stage approach that combines design time mapping and DVFS strategy with run-time thermal optimization. In design time mapping, tasks are mapped to data flow graphs, e.g., Directed Acyclic Graphs (DAGs) [12]. Based on a complete power model with leakage power effect, the method first assigns tasks' data flow graphs (DAGs) to suitable layers of a 3D-MPSoC to achieve power balance and meet the throughput constraints. The available slack for tasks is exploited to perform DVSF for further peak temperature reduction. Following the design stage, a run-time thermal optimization strategy is applied using runtime information to exchange the workloads from hot core stacks to cool stacks without affecting the periodic constraint of the application. This is the solution that addressed thermal-aware mapping on 3D-MPSoC while still considering DVFS enabled cores and leakage power effects. However, mapping tasks into data flow graphs (DAGs) would be complicated in real applications because the number of tasks could be very large and the number of active tasks could change dynamically during runtime.

Furthermore, Chien-Hui et al. [11] proposed a new thermal-constrained task scheduler based on a thermal-pattern-aware voltage assignment (TPAVA) for 3D-MPSoCs. It consists of an online allocation strategy and a new voltage-scaling strategy to maximize throughput and minimize occurrences of hotspots. First, the thermal behaviour of 3D-MPSoCs under different voltage levels is analysed to generate a temperature profile of all cores under different voltage levels. According to these temperature profiles, TPAVA pre-emptively assigns different operating-voltage levels to all cores for reducing the temperature in 3D-MPSoCs during runtime. In addition, by considering the strong thermal correlation among vertically-aligned cores, a new vertical-grouping voltage scaling (VGVS) strategy is proposed for cooling 3D-MPSoCs effectively. The VGVS suggests Super-Task-to-Super-Core (STSC) strategy, which defines super tasks (tasks in the same vertical cores) and super cores. (It groups vertically-aligned cores as a super core). STSC allocates a super task, i.e., set of tasks, to a super core. Therefore, the three-dimensional allocation problem can be reduced to two-dimensional allocation problem by the STSC strategy. Moreover, VGVS strategy triggers DVFS on the overheated core as well as its vertically-aligned cores. Thermal behaviours on vertically-aligned cores and core-to-heat-sink distance are utilized by the online allocation strategy, i.e., vertical-grouping voltage scaling (VGVS) strategy to reduce hotspots in 3D-MPSoCs. The drawback of this solution is the overhead of vertical-grouping and voltage scaling strategy. If the task migration happens frequently in vertically-aligned cores, it may introduce a large overhead that needs to be considered.

Although many research studies have been proposed to deal with thermal adversity in 3D-MPSoCs, none of those solutions have considered the existing cache hierarchies and their suitability for use in 3D-MPSoCs. There are some cache hierarchies and cache policies that may not suite 3D-MPSoCs in terms of performance and energy efficiency. Previously, several studies have shown that caches can significantly improve the performance of a system, but it consumes a large amount of energy [13]. A few works have reported that the caches account for as much as 50% of the total energy consumption [2,3,14]. Hence, selecting a suitable cache hierarchy and cache policy in 3D-MPSoCs is even more important.

Chapter 4

Experimental Setup

We conduct our study by the experimental method. We consider an existing multicore processor as a layer of processor cores in a 3D-MPSoC. We perform experiments on two typical multicore architectures: a heterogeneous multicore system with exclusive cache hierarchy and a homogeneous multicore system with inclusive cache hierarchy. The purpose is to understand how these different cache hierarchies could be a problem for multi-threaded applications in multicore systems and how data sharing through cache could be benefited/obstructed by these cache hierarchies. Hence, analysing the hurdles of data sharing in cache hierarchies in existing multicore systems would help to understand their suitability for use in 3D-MPSoCs. In this chapter, we first present two hardware architectures: Odroid-XU4 for heterogeneous cores with exclusive cache [15] and Intel NUC for homogeneous cores with inclusive cache [16]. The section after that presents the necessary tools and setup for experimental purposes.

4.1 Hardware Platforms

4.1.1 Odroid-XU4 Architecture

Odroid-XU4 is a heterogeneous multicore processor with big.LITTLE architecture [17] designed for high performance and energy efficiency. Big.LITTLE technology uses two types of processor core: 'LITTLE' processor core designed for maximum power efficiency, and 'big' processor core designed for maximum computing performance. Both types of processor core are coherent and share the same instruction set architecture. Using big.LITTLE technology, each task can be dynamically allocated to a big or

LITTLE core depending on the performance requirement of that task. Figure 4.1 shows the overall architecture of the Odroid-XU4 with four LITTLE cores ARM Cortex A7 (each core runs at 1.4 GHz) and four big cores ARM Cortex A15 (each core runs at 2.0 GHz).

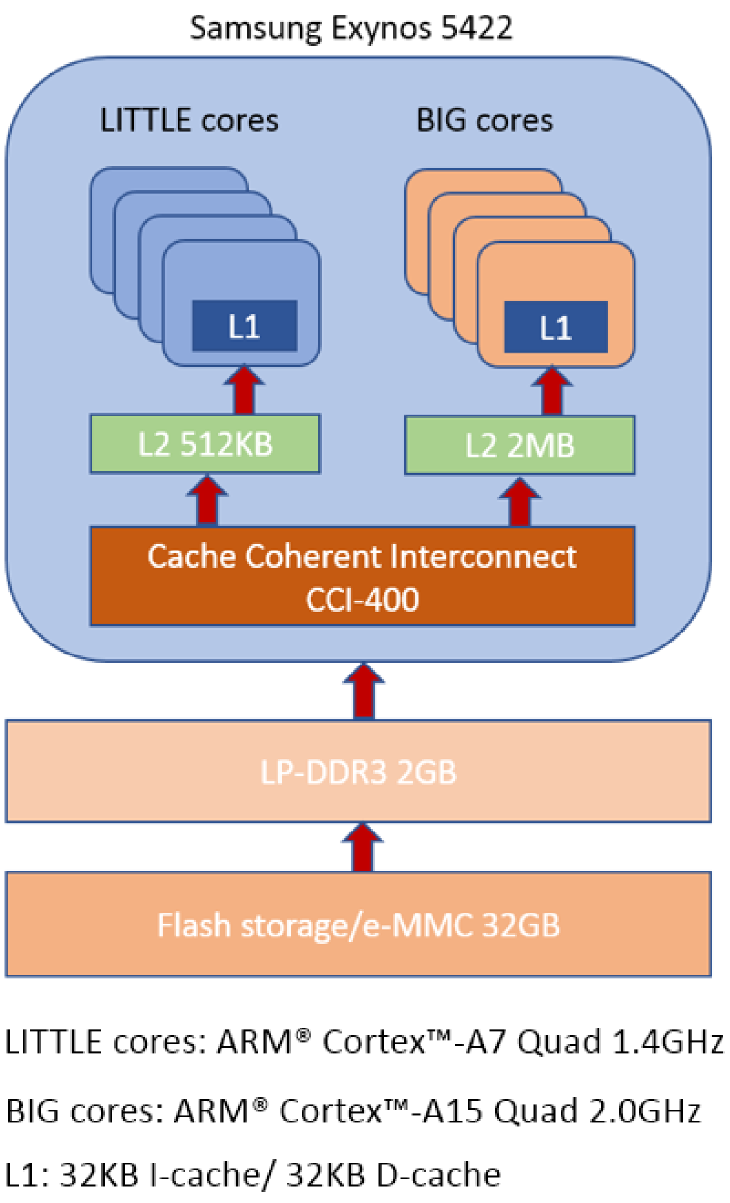
Samsung Exynos 5422
LITTLE cores
BIG cores
L1
L1
L2 512KB
L2 2MB
Cache Coherent Interconnect CCI-400
LP-DDR3 2GB
Flash storage/e-MMC 32GB
LITTLE cores: ARM® Cortex™-A7 Quad 1.4GHz
BIG cores: ARM® Cortex™-A15 Quad 2.0GHz
L1: 32KB I-cache/ 32KB D-cache

Figure 4.1: Odroid XU4 architecture.

The memory system consists of two-level cache memory, cache coherent interconnect, external memory (LP-DDR3) and external data storage (eMMC). In the LITTLE cores, each L1 cache is 32KB I-cache (instruction cache) and 32KB D-cache (data cache). The four LITTLE cores share the same L2 cache of 512KB. Similarly, in the big cores, each L1 cache is 32KB I-cache and 32KB D-cache. The four big cores share an L2 cache of 2MB which is larger than the L2 cache of LITTLE cores. The two clusters (a group of identical processor cores) communicate via special hardware connection called Cache Coherent Interconnect (CCI-400) which connects the two shared L2 caches. [17] explains in detail about this CCI-400 which enables the seamless data transfer between two clusters. Without CCI, the transfer of data between big and LITTLE cores would always occur through the main memory, and this would be slow and power inefficient.

4.1.2 Intel NUC Architecture

Intel NUC is a homogeneous multicore architecture. The processor has four identical cores running at the same speed of 1.6GHz. The memory subsystem is inclusive cache hierarchy with four L1 private caches and a shared L2 cache.

Figure 4.2 shows a general architecture of Intel NUC and its memory system hierarchy. Intel NUC uses four processor cores N3700 running at 1.6GHz. Each processor core has a private L1 cache composed of 24KB D-cache and 32KB I-cache. The L2 cache is shared among four cores with the capacity of 1024KB.

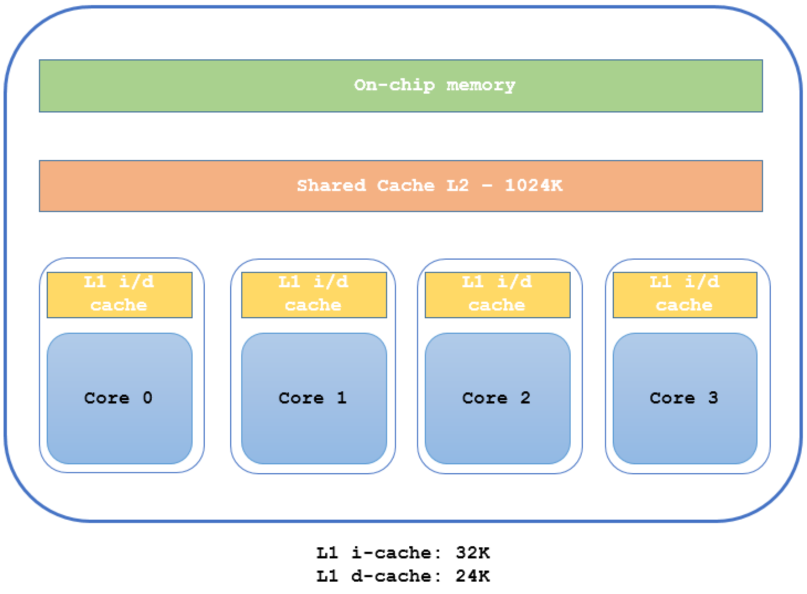

Figure 4.2: Intel NUC architecture

4.2 Experimental Setup

The purpose of this experimental setup is to examine how the existing heterogeneous/homogeneous multicore architectures with exclusive/inclusive cache hierarchies could help a multi-thread application to speed up its execution by enabling the data sharing through cache memory. To be specific, we compare the execution time of a single-thread application with a parallel-thread application. The parallel-thread application is created from the original single-thread application by spawning another thread which is identical to the main thread of the original application. Logically, the parallel-thread application should execute faster than the original single-thread application because the second thread which acts as a helper thread would help to prefetch requested data into the shared cache memory first. Hence, the total execution time of the parallel-thread application is reduced due to the availability of processing data in the shared cache. This experimental method helps us to understand the obstacles or benefits of data sharing through shared cache memory in available multicore systems and its suitability for use in 3D-MPSoCs.

- First, both Odroid-XU4 and Intel NUC system are set up to run Ubuntu Mate 16.04, Linux kernel version 4.9.27
- Next, we need a tool to isolate two processor cores on these platforms to run our parallel-thread/single-thread applications so that they are not affected by OS scheduler which may migrate task/thread to any other random cores during its execution. CPUSet tool is used to achieve this.
- Lastly, a few other tools, such as 'htop', 'pstree' are used to observe the process's ID (PID) and processor workload during experiments.

4.2.1 CpuSet

Cpuset is an important affinity tool in Linux system to provide a mechanism to assign a set of processor cores to a set of tasks. To install **cpuset**

```
$sudo apt-get install cpuset
```

Listing 4.1 shows some examples of using cpuset to shield certain processor cores. A shield is a protection layer that isolates a set of processor cores from external interrupts, OS scheduler and system tasks. The following examples explain how to tear down a shield after use, to list all tasks in a shield, and to move a task in and out a shield.

```
Shield CPU 1-3, CPU0 become system CPU
$sudo cset shield --cpu 1-3

Tear down a shield
$sudo cset shield --reset

List #tasks in the shield
$sudo cset shield --shield -v

Listing all cpuset
$sudo cset set --list

List tasks in a cpuset
$sudo cset proc --list --set=user --verbose

Moving a task
$sudo cset proc --toset=user --move --pid xxxx,xxxx,xxxx

Moving a task and all its siblings
$cset proc --move --toset=user --pid xxxx --threads

Moving all tasks from one cpuset to another
$cset proc --move --fromset=system --toset=user
```

Listing 4.1: cpuset example

4.2.2 MiBench Applications

Previous section prepared for the platform setup. In this section, we prepare the applications for our experimental purpose. We select MiBench for our benchmark test suite because it supports different applications with various input data type. The benchmark applications are selected and modified with general rule as follows:

- The main loop of an application is put into iterations to increase the total execution time for easy comparison with the execution time of the parallel-thread application. The original application usually only executes in a few seconds so that by increasing its execution time to 2-3 minutes it is easy to see the difference compared to the execution time of the parallel-thread application.

- The input data size is increased to magnify the effect of data sharing in shared cache memory so that we can evaluate which cache hierarchy benefits or obstructs the performance of data sharing in multicore systems.

- The *printf* code is carefully removed to avoid interference with our evaluation during the code's execution.

- The parallel-thread application is created from a single-thread application with the help of our wrapper code. This wrapper code is written in Python, and it invokes the single-thread application to spawn another identical thread. Further details of the wrapper code will be presented in the next chapter.

Example: Below is an example to demonstrate the procedure of modifying a MiBench application. We use "dijkstra" application for the illustration purpose.

1. Input file size is increased to 6 times.

2. *Printf* is carefully removed from the original code to avoid interactive delay during the code execution.

3. Modify the *main()* function in *dijkstra_large.c* to repeat the iteration. For example, in the below snippet of code **NUM_NODES** is set to 3000 (originally it was 100).

```
#define NUM_NODES    3000

int main(int argc, char *argv[]) {
  int i,j,k;
  FILE *fp;

  if (argc<2) {
    //fprintf(stderr, "Usage: dijkstra <filename>\n");
    //fprintf(stderr, "Only supports matrix size is #define'd.\n");
  }
  /* open the adjacency matrix file */
  fp = fopen (argv[1],"r");

  /* make a fully connected matrix */
  for (i=0;i<NUM_NODES;i++) {
    for (j=0;j<NUM_NODES;j++) {
      /* make it more sparce */
      fscanf(fp,"%d",&k);
                              AdjMatrix[i][j]= k;
    }
  }

  /* finds 10 shortest paths between nodes */
  for (i=0,j=NUM_NODES/2;i<100;i++,j++) {
                              j=j%NUM_NODES;
      dijkstra(i,j);
  }
  exit(0);

}
```

Listing 4.2: Example of modify *dijkstra* application

We select five applications from MiBench and modify as per the previous example. Each application has different input data type and categorized into different application segments. The following five applications are selected for experiments:

- Blowfish: a cipher algorithm that implements data encryption and decryption.
- Dijkstra: an algorithm that finds the shortest paths between nodes in a graph.
- Qsort: an algorithm for sorting list of 3D vectors based on distance from the origin.
- Sha: a secure hash algorithm to generate a unique ID for any block of data.
- Susan: a method for digitally processing images to determine the position of edges and corners therein for the guidance of unmanned vehicle.

In the next chapter, we will present the experiment method and results on both Odroid-XU4 and Intel NUC platforms.

Chapter 5

Experimental Method

This chapter presents the experimental method to examine the performance of data sharing in the shared cache memory by multi-thread applications in a multicore system. We select five applications from MiBench which originally were single-thread applications. The wrapper code is developed to create a parallel-thread application from the original single-thread application. The two threads are identical, process the same input data and run on two different cores. During the execution of an application, one of the thread acts as a helper thread to prefetch the required data into the shared L2 cache. Hence, the other thread will process data faster due to the availability of requested data in the cache memory, and that saves the time to load data from external memory into the local cache memory.

5.1 Experimental Method

We perform the experiment on Odroid-XU4 and Intel NUC platforms. On each platform, we select two processor cores for the experiment purpose. These processor cores are isolated from Linux scheduler by using "cset" tool to create a shield. The shield prevents other system tasks to interfere with the experiment. Under the shield, we execute the original thread and its identical thread in parallel but in two different processing cores. By doing so, necessary data from main memory is prefetched into the last level of the shared cache memory (LLC). The data required by the actual thread is being cached earlier by the other core, and therefore, the execution time of the actual task is reduced. In other words, our experiment principle should speed up the execution of the main thread and improve the cache-miss rate in the last level cache (LLC). Figure 5.1 explains our experiment principle.

On the other hand, we execute the original single-thread applications in the same shield and measure the execution time. The execution time of single-thread applications is compared with execution time of parallel-thread applications to examine if the parallel-thread applications could speed up the execution time due to the help of the prefetch thread and the availability of requested data in the shared cache memory.

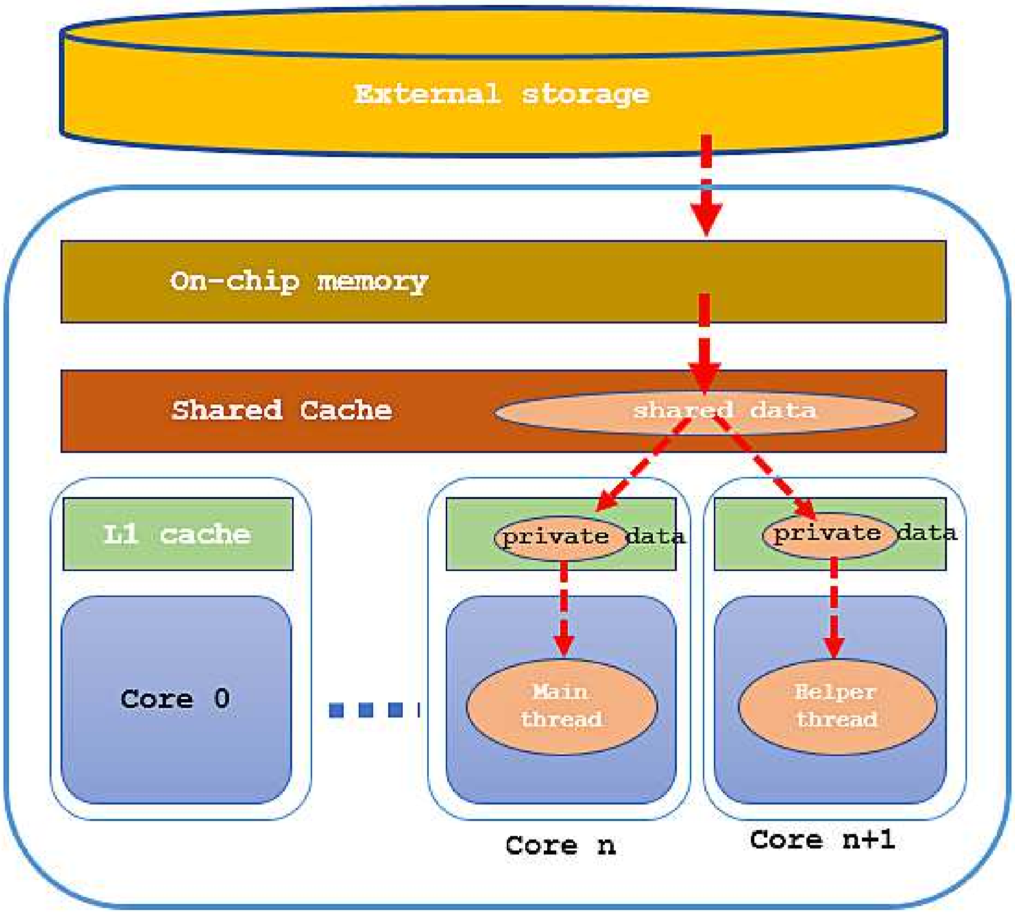

Figure 5.1: Experimental Method

5.2 The Wrapper Code

To facilitate our experiment, we developed a generic wrapper code in Python. The wrapper code takes input as an application name and parallelizes the original application into two threads. The two identical threads will run on two different cores.

The wrapper code from listing 5.1 can be explained as follows:

The wrapper code first defines two threads that run in parallel. It also defines the semaphore number that limits the number of threads. We only need two threads in our experiment; hence the number is two. The next function defines function *run_command* to use in the main loop of the wrapper code. It uses *os.system()* to call the application from the input argument. In the main loop, the two threads are created by using Python threading library (*threading.Thread()*). Name of the executable application is passed into the *Thread()* function so that the two threads are created identically, and they will start to run concurrently.

```
#!/usr/bin/python

import threading
import os

#Semaphore limits the number of threads in parallel

semaphore = threading.Semaphore(2)

def run_command(cmd):
    with semaphore:
        os.system(cmd)

#Each loop starts a thread
for i in range(2):
        threading.Thread(target=run_command, args=("./patricia large.udp
    ",)).start()

#To execute: ./parallel_command.py
```

Listing 5.1: Wrapper code *parallel_command.py*

Chapter 6

Experimental Results

This chapter presents the experimental results on two typical multicore systems: heterogeneous and homogeneous architectures. Heterogeneous architectures normally deploy with exclusive cache hierarchies. Due to the difference in clock frequency in heterogeneous processor cores, heterogeneous systems need a special hardware to bridge the data communication between the different clusters of cores and to ensure the cache coherency (such as Cache Coherent Interconnect, aka CCI). Hence the cache hierarchy in heterogeneous systems seems to be complex with two layers of caches and another layer of CCI. On the other hand, homogeneous systems normally deploy with inclusive cache hierarchies. All processor cores in a homogeneous system run at the same clock frequency and do not need the special hardware such as CCI. Cache hierarchies in homogeneous systems are multiple levels such as two or three levels of cache architecture. In this work, we present experimental results on Odroid-XU4 platform and Intel NUC platform. Odroid-XU4 platform represents for heterogeneous multicore systems with exclusive cache hierarchy. On the other side, Intel NUC platform represents a homogeneous multicore system with an inclusive cache hierarchy. The experiments are performed to compare the execution time of parallel-thread application with the original single-thread application to understand how existing heterogeneous/homogeneous architectures and exclusive/inclusive cache hierarchies may help or affect data sharing in a shared cache memory. We consider an existing multicore processor as a layer of processor cores in 3D-MPSoCs. Hence understand the hurdles in data sharing in existing multicore systems could help to understand the performance hurdles in 3D-MPSoCs. Data sharing is an important feature of 3D-MPSoCs to accelerate multi-threading and multicore processing.

6.1 Experimental Results on Odroid-XU4

We run five MiBench applications on different configurations on Odroid-XU4. The execution time is measured with both ordinary execution (single thread) and parallel-thread execution. The different configurations are:

- Configuration 1: threads run on two 'LITTLE' cores
- Configuration 2: threads run on two 'big' cores
- Configuration 3: threads run on a pair of 'big-LITTLE' cores

We noticed that in Odroid system, the 'LITTLE' cores are numbered from 0 to 3, and 'big' cores are numbered from 4 to 7. This information can be obtained from the system using command '*lscpu*'.

6.1.1 Experimental results on two LITTLE cores

This section presents our experimental results on the two 'LITTLE' cores of Odroid-XU4. The experiment is performed in the following steps:

- We first create a shield with two LITTLE cores for the experiment.
- The single-thread applications are executed in this shield and the execution time is measured.
- The parallel thread applications are executed in the same shield and the execution time is measured.

Figure 6.1 presents the configuration for the experiment on two 'LITTLE' cores

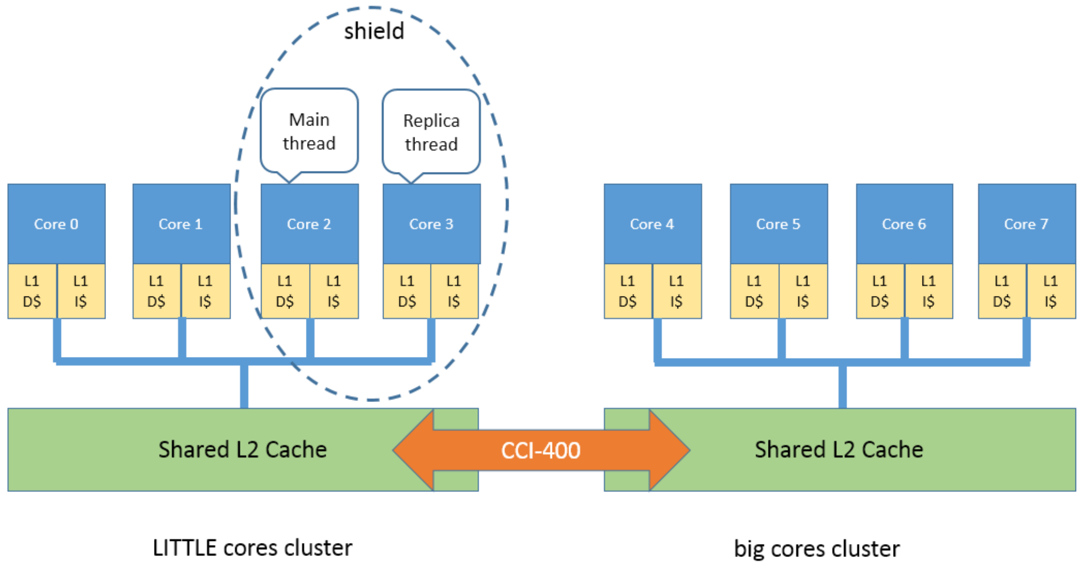

Figure 6.1: Experiment on two LITTLE cores

Listing 6.1 below shows our experiment steps with the 'blowfish' application executing on two 'LITTLE' cores 2, 3.

We first run the cpu tool '*cset*' to create a shield for cores 2, 3 which are the two 'LITTLE' cores. The single-thread application is executed by the script '*runme_large.sh*'. The script is then executed with the command '*time*' to measure the execution time and command '*cset shield –exec*' to run the application within the shield. The '*time*' command measures the execution time in three outputs: real time, user time and system time. The total execution time of the application is the result of the 'real time', which is 0m55.093s.

Similarly, the parallel-thread application is executed by the script '*parallel_command.py*' in the same shield and the execution time is measured with '*time*' command. Because the shield is created with two processor cores, each thread will be run on a physical core. The execution time of the parallel-thread application in this example is 1m2.326s.

```
root@odroid:/home/odroid# cset shield -c 2,3
cset: --> activating shielding:
cset: moving 112 tasks from root into system cpuset...
[==================================================]%
cset: "system" cpuset of CPUSPEC(0-1,4-7) with 112 tasks running
```

```
cset: "user" cpuset of CPUSPEC(2-3) with 0 tasks running

#blowfish
odroid@odroid:~/MibenchModified/blowfish_modified$ time cset shield --
    exec ./runme_large.sh
cset: **> 1 tasks are not movable, impossible to move
cset: --> last message, executed args into cpuset "/user", new pid is:
    2029

real    0m55.093s
user    0m54.530s
sys     0m0.540s

odroid@odroid:~/MibenchModified/blowfish_modified$ time cset shield --
    exec ./parallel_command.py
cset: **> 1 tasks are not movable, impossible to move
cset: --> last message, executed args into cpuset "/user", new pid is:
    2037
real    1m2.326s
user    2m3.085s
sys     0m1.320s
```

Listing 6.1: Experiment example in two LITTLE cores

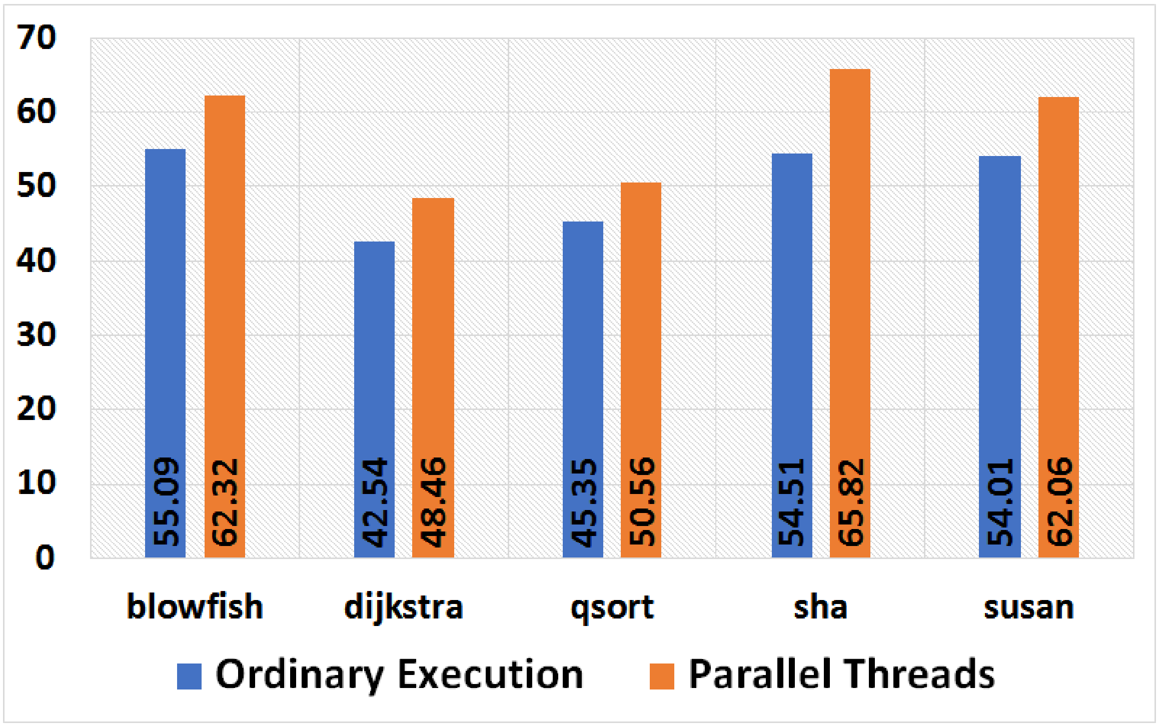

Figure 6.2: Execution time comparison when threads run in two LITTLE cores.

Figure 6.2 shows the execution time of parallel-thread applications compared with single-thread applications when threads run in two LITTLE cores. Results are collected from the experiment with five MiBench applications. The execution time is recorded in seconds. In this experiment, the execution time of parallel-thread applications is longer than the execution time of single-thread applications. The parallel threads and the shared cache memory do not improve application's performance in this scenario.

6.1.2 Experimental results on two big cores

This section presents our experimental results on the two 'big' cores of Odroid-XU4. The experiment is performed in the following steps:

- We first create a shield with two 'big' cores for the experiment.
- The single-thread applications are executed in the shield and the execution time is measured.
- The parallel-thread applications are executed in the shield and the execution time is measured.

Figure 6.3 presents the configuration for the experiment on two ‘big’ cores

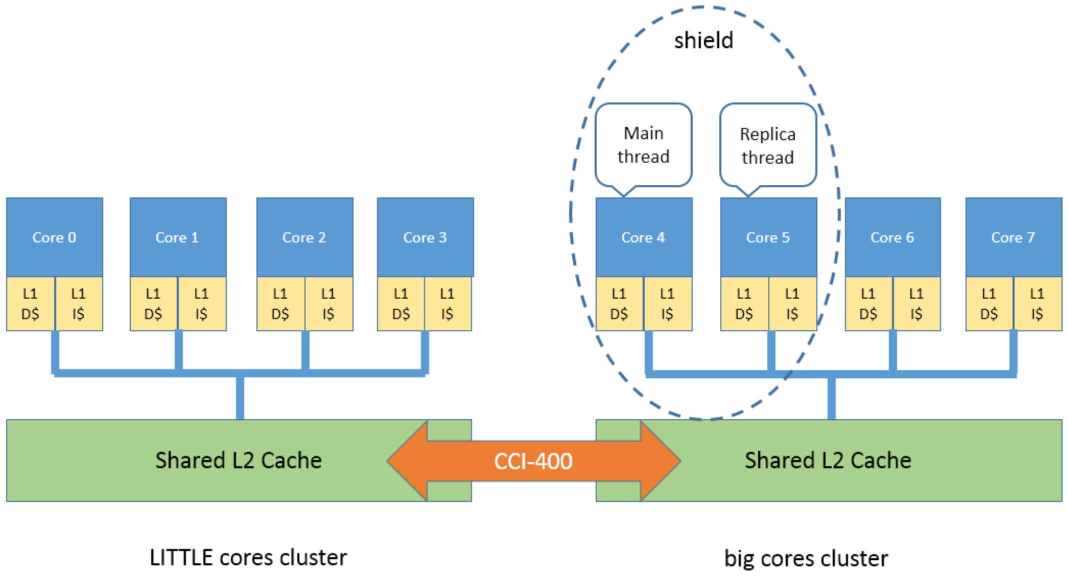

Figure 6.3: Experiment on two big cores.

Listing 6.2 below shows our experiment steps with the ‘dijkstra’ application executing on two ‘big’ cores 4, 5.

We first run the cpu tool ‘*cset*’ to create a shield for cores 4, 5 which are the two ‘big’ cores. The experiment steps are performed similarly to the procedures in two ‘LITTLE’ cores. The total execution time of the single-thread application is 0m42.958s

Similarly, the parallel-thread application is run by the script ‘*parallel_command.py*’ in the same shield and the execution time is measured with ‘*time*’ command. The execution time of parallel-thread application in this example is 0m47.426s.

```
root@odroid:/home/odroid# cset shield --c 4,5
cset: --> activating shielding:
cset: moving 111 tasks from root into system cpuset...
[==================================================]%
cset: "system" cpuset of CPUSPEC(0-3,6-7) with 111 tasks running
cset: "user" cpuset of CPUSPEC(4-5) with 0 tasks running

#dijkstra
root@odroid:/home/odroid/MibenchModified/dijkstra_modified# time cset
    shield --exec ./runme_large.sh
```

```
cset: --> last message, executed args into cpuset "/user", new pid is:
    1390

real    0m42.958s
user    0m35.040s
sys     0m7.905s
root@odroid:/home/odroid/MibenchModified/dijkstra_modified# time cset
    shield --exec ./parallel_command.py
cset: --> last message, executed args into cpuset "/user", new pid is:
    1392

real    0m47.426s
user    1m15.730s
sys     0m17.965s
```

Listing 6.2: Experiment example in two big cores

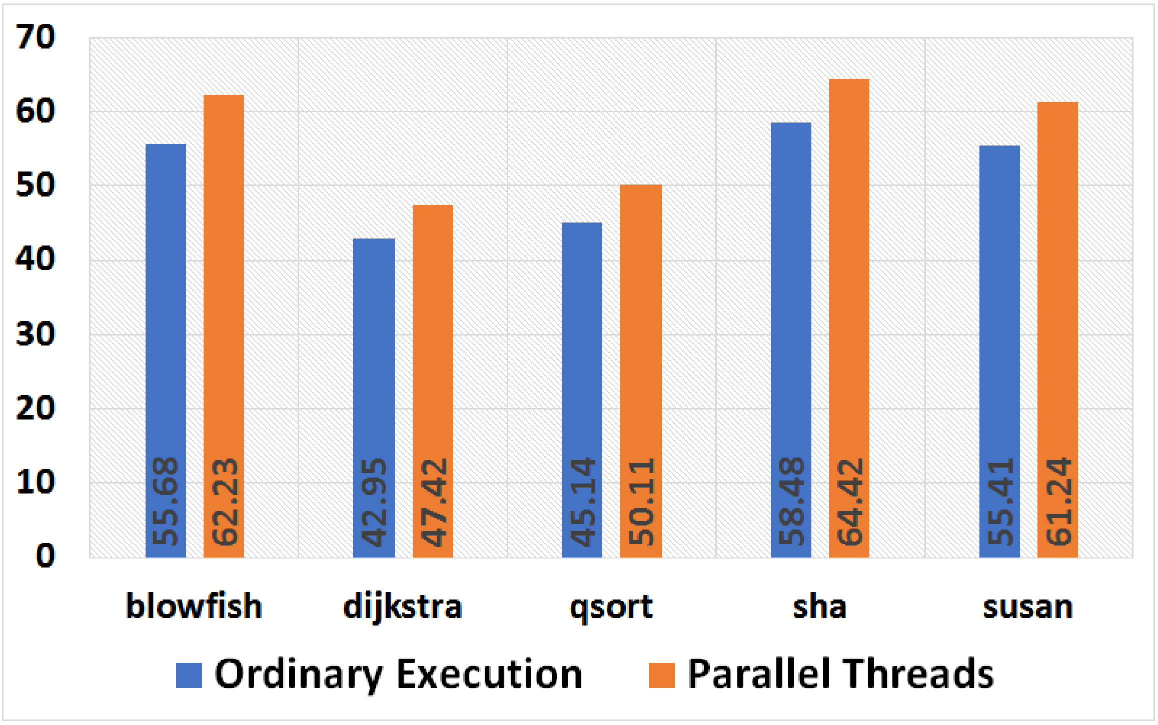

Figure 6.4: Execution time comparison when threads run in two big cores.

Figure 6.4 shows the execution time of parallel-thread applications compared with single-thread applications in two big cores. Results are collected from the experiment with five MiBench applications. The execution time is recorded in seconds. In this experiment, the execution time of parallel-thread applications is longer than the execution time of single-thread applications. The parallel threads and the shared cache memory do not improve application's performance in this scenario.

6.1.3 Experimental results on a pair of big-LITTLE cores

This section presents our experimental results on a pair of big-LITTLE cores. The experiments procedures are performed similarly to previous cases.

Figure 6.5 presents the configuration for the experiment on a pair of big-LITTLE cores 3, 4.

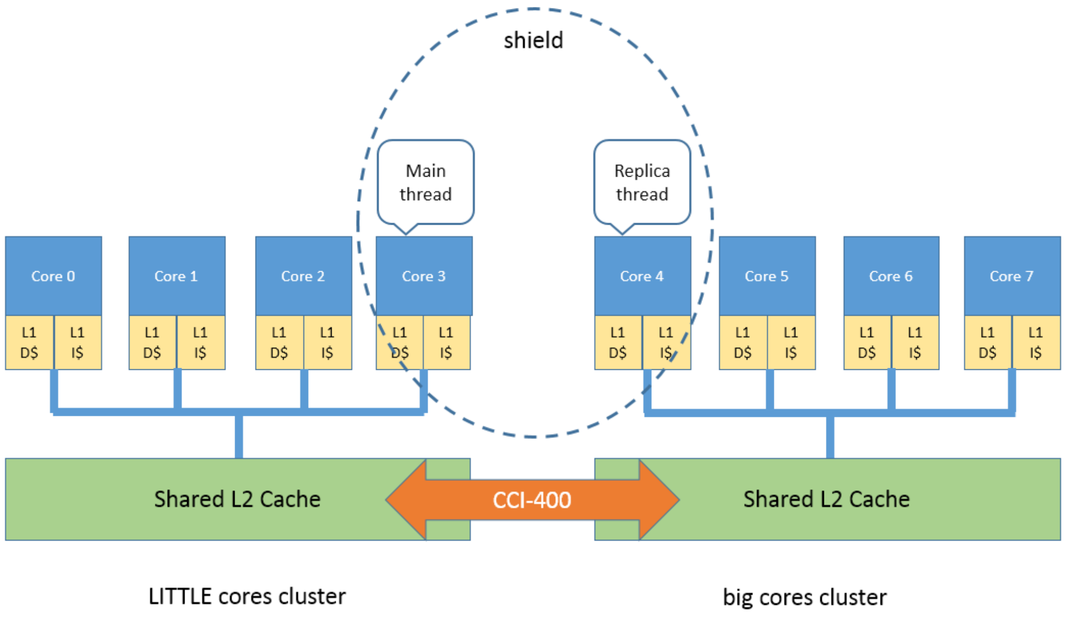

Figure 6.5: Experiment on a pair of big-LITTLE cores

Listing 6.3 below shows our experiment steps with the 'qsort' application executing on a pair of big-LITTLE cores. The total execution time of the single-thread application in this example is 1m5.751s. The execution time of parallel-thread application in this example is 1m27.414s.

```
root@odroid:/home/odroid# cset shield -c 3,4
cset: --> activating shielding:
cset: moving 109 tasks from root into system cpuset...
[==================================================]%
cset: "system" cpuset of CPUSPEC(0-2,5-7) with 109 tasks running
cset: "user" cpuset of CPUSPEC(3-4) with 0 tasks running
```

```
#qsort
root@odroid:/home/odroid/MibenchModified/qsort_modified# time cset
    shield --exec ./runme_large.sh
cset: --> last message, executed args into cpuset "/user", new pid is:
    1406

real    1m5.751s
user    0m45.360s
sys     0m20.375s
root@odroid:/home/odroid/MibenchModified/qsort_modified# time cset
    shield --exec ./parallel_command.py
cset: --> last message, executed args into cpuset "/user", new pid is:
    1409

real    1m27.414s
user    0m53.590s
sys     0m33.780s
```

Listing 6.3: Experiment example in a pair of big-LITTLE cores

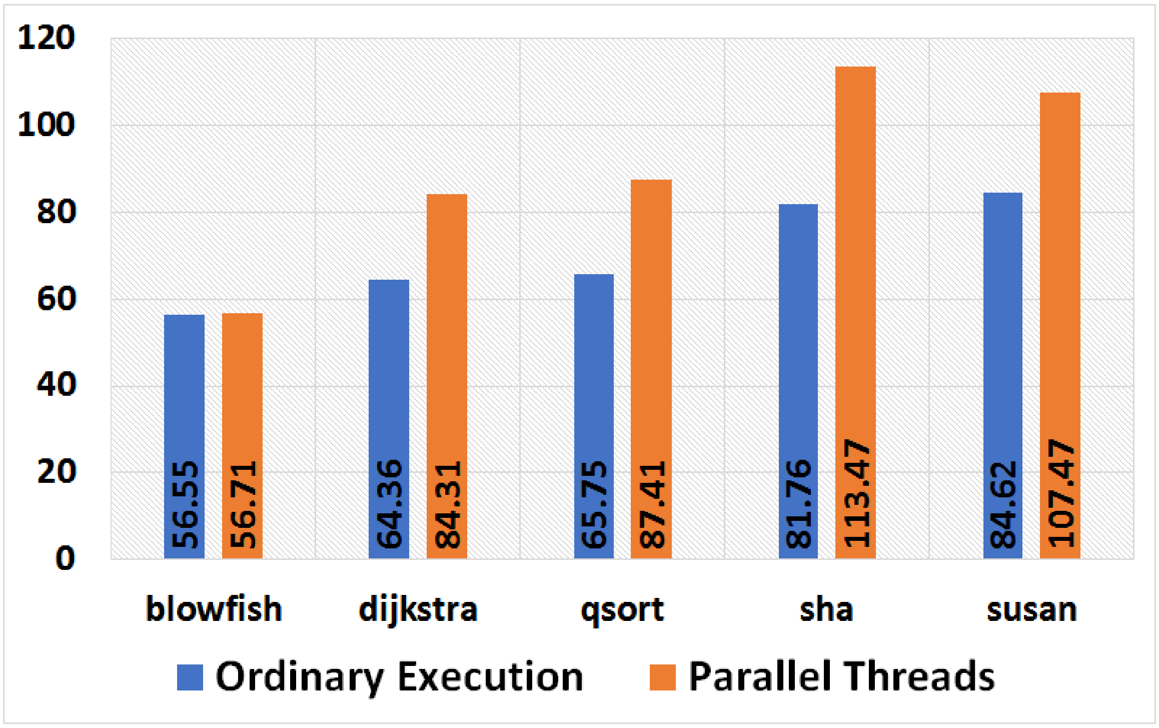

Figure 6.6: Execution time comparison when threads run in a pair of big-LITTLE cores.

Figure 6.6 shows the execution time of parallel-thread applications compared with single-thread applications in a pair of big-LITTLE cores. Results are collected from the experiment with five MiBench applications. The execution time is recorded in seconds. In this experiment, the execution time of parallel-thread applications is longer than the execution time of single-thread applications. The parallel threads and the shared cache memory do not improve application's performance in this scenario.

6.1.4 Results Analysis on Odroid-XU4

From the experimental results, we see that parallel threads with the shared cache memory in Odroid-XU4 do not improve applications' performance in all scenarios. When the two threads are on the same cluster (LITTLE cores or big cores), the application's performance is degraded mostly at 15% total execution time compared to the single-thread executions. When the main thread and its replica thread are on two different clusters (one thread is on a big core, and the other thread is on a LITTLE core), the application's performance is degraded mostly at 30% total execution time compared to the single-thread execution.

The experimental results on Odroid-XU4 explain that existing heterogeneous architecture and exclusive cache hierarchy do not help for data sharing in the shared cache memory. Especially with multi-threading and multicore processing, the existing cache hierarchy in heterogeneous system shows the obstacle in improving the execution time and performance of multi-thread applications in multicore systems.

Furthermore, we analyse the memory subsystem of Odroid, and we understand that Odroid uses exclusive cache hierarchy. It also uses special hardware such as Cache Coherent Interconnect (CCI) to ensure the cache coherency between heterogeneous cores. Hence, we draw the conclusions as follows:

- Odroid uses exclusive cache hierarchy that creates trouble in data sharing among multiple cores. In exclusive cache hierarchy, a data block can be found in either an L1 cache or the L2 cache, but not in both. Upon an L1 miss, other L1 caches and L2 will have to be looked up. The lookup method is implemented by cache snooping-based protocol. However, with snooping-based protocols, the search operation becomes complicated with more overhead when the number of cores is increased. The search operation is performed in entire cache space. Hence, it takes more time to search for the requested data block.

- CCI (Cache Coherent Interconnect) in heterogeneous systems is supposed to help in data coherency, but it introduces extra overhead by synchronizing data between heterogeneous cores. The CCI creates more delay in execution time such as a processor core has to wait for the requested data to be synchronized from the other cluster. That explains the experimental results when threads run on two different clusters, the execution time of parallel-thread application is degraded mostly at 30% compared to the execution time of the single-thread execution.

6.2 Experimental Results on Intel NUC

In the previous section, we performed various experimental works on Odroid system. We concluded that the heterogeneous multicore system and exclusive cache hierarchy are troublesome for data sharing for multi-threaded applications in multicore systems. We continue to expand the experiments on another architecture which is a homogeneous system and inclusive cache hierarchy to further analyse the performance of data sharing in the shared cache memory in this architecture. We choose Intel NUC platform for this experiment because this is a homogeneous multicore system with four processor cores and its memory subsystem uses inclusive cache hierarchy. Intel NUC is a homogeneous architecture so that all processor cores are identical. In this scenario, we do not have many configurations for the experiment as we did previously in Odroid system. Instead, we only have one configuration for the experiment in Intel NUC. We select any two cores of the system to perform our experiment. In figure 6.7, we select core 2, 3 to place in a shield for the experiment.

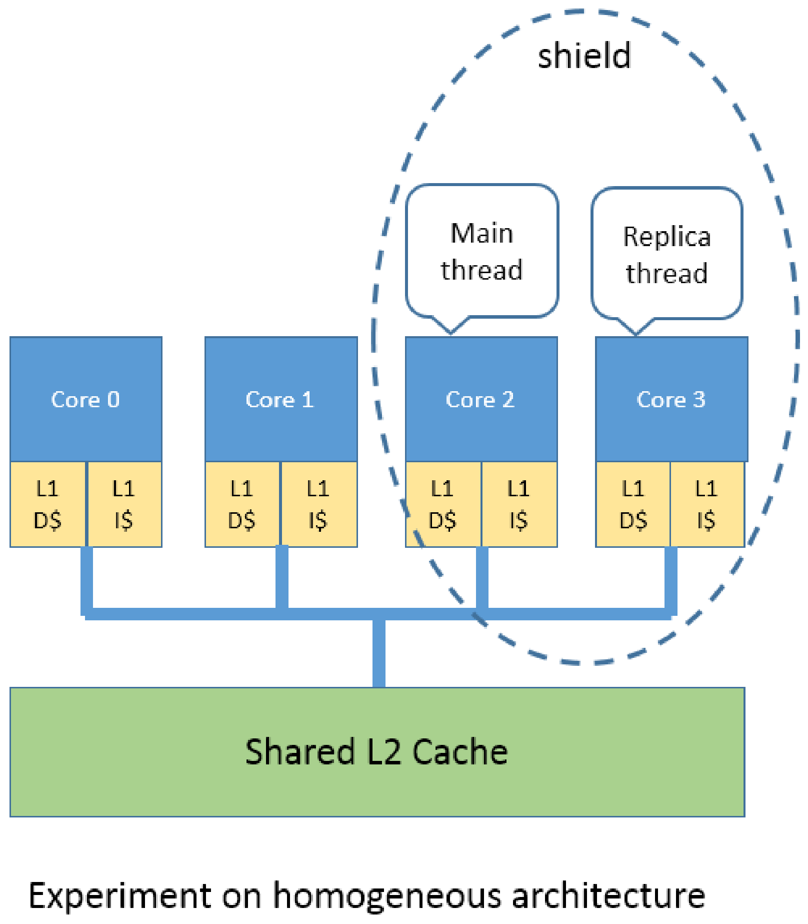

Figure 6.7: Experiment on two cores of homogeneous architecture

We use the five applications from MiBench for our experiment on Intel NUC. The five applications are *blowfish, dijkstra, qsort, sha, susan.* The experiment steps are done similarly to what we did on Odroid system.

- We first create a shield with two cores for the experiment (for example, core 2,3).
- We run the single-thread applications in the shield and measure the execution time.
- We run the parallel-thread applications in the shield and measure the execution time. (Each thread runs on a separated core)
- Finally, we compare the execution time of the parallel-thread application with its correspondent single-thread application to see if the replica thread and data

sharing in shared cache memory could help to speed up the application execution.

- In this experiment, we use perf tool to measure the execution time of applications. Moreover, perf is able to measure the cache misses in LLC. Due to 'perf' being a hardware-dependent tool, we can use perf only on Intel platform but not on Odroid or other ARM-based platforms.

Below listing 6.4 shows an example to run the experiment with the application '*dijkstra*' in the shield.

```
# cset shield -c 2,3
cset: --> activating shielding:
cset: moving 344 tasks from root into system cpuset...
[==================================================]%
cset: "system" cpuset of CPUSPEC(0-1) with 344 tasks running
cset: "user" cpuset of CPUSPEC(2-3) with 0 tasks running

# perf stat -e cache-references,cache-misses cset shield --exec ./
    parallel_command.py
cset: --> last message, executed args into cpuset "/user", new pid is:
    8584

 Performance counter stats for 'cset shield --exec ./parallel_command.py
    ':


    18,921,929,327      cache-references
       779,203,669      cache-misses              #    4.118 % of all
    cache refs


     263.342451404 seconds time elapsed

# perf stat -e cache-references,cache-misses cset shield --exec ./
    runme_large.sh
cset: --> last message, executed args into cpuset "/user", new pid is:
    8549

 Performance counter stats for 'cset shield --exec ./runme_large.sh':
  10,369,663,514      cache-references
     520,790,448      cache-misses                  #    5.022 % of all
 cache refs

   330.717180611 seconds time elapsed
```

Listing 6.4: Running an application with *perf* and *cset*

In listing 6.4, the shield is first created with core 2, 3. The parallel-thread is created by calling the wrapper code '*parallel_command.py*'. The specific application (dijkstra) is called inside the wrapper code by changing the application name in the script. The *perf* tool is used to measure the execution time. It also measures cache misses by using the parameters '*perf stat -e cache-references, cache-misses*'. '*Cache-references*' is the total cache hits of the application's execution, and '*cache-misses*' is the total cache misses of the application's execution. The above example shows the execution time of a parallel-thread application (*dijkstra*) is 263.342s with cache-misses is 4.118% of all cache references. The execution time of the single-thread application (*dijkstra*) is 330.717s with cache-misses is 5.022% of all cache references.

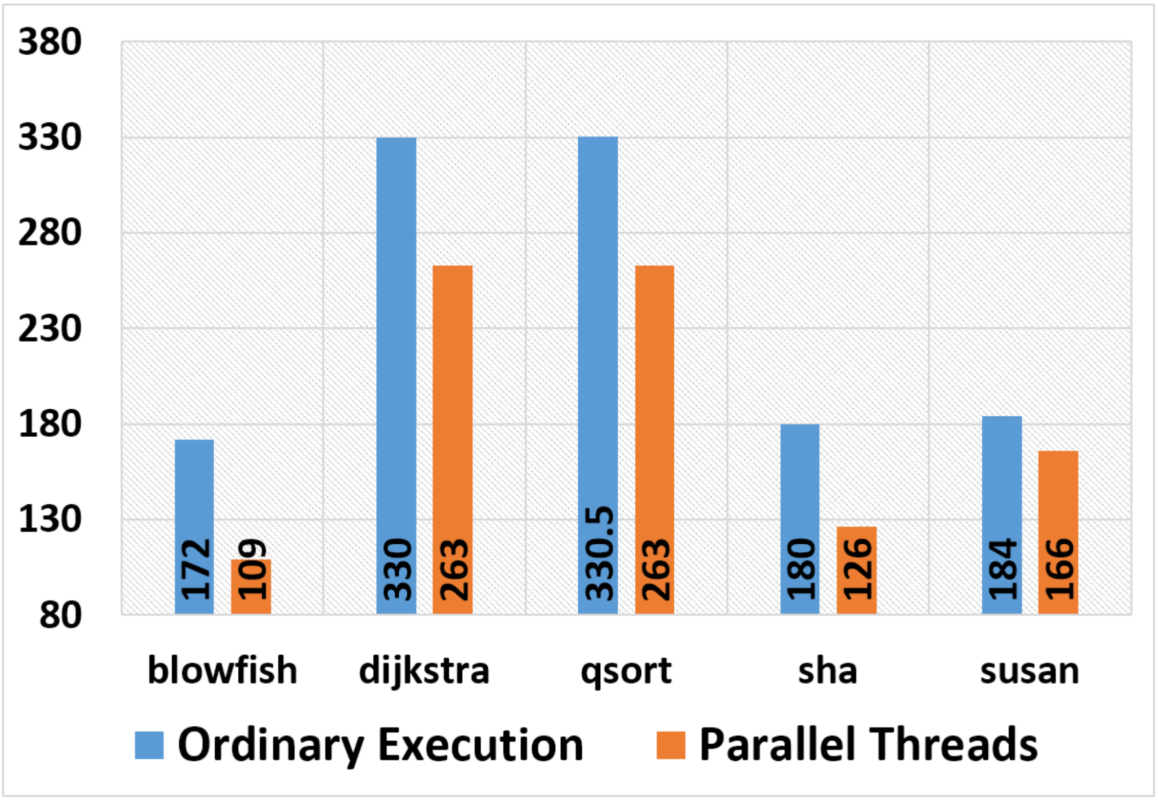

Figure 6.8: Execution time comparison when threads run in two cores of Intel NUC.

Figure 6.8 presents the experimental results on Intel NUC platform with five applications from MiBench. Execution time is measured in seconds. The results show that parallel-thread applications speed up the execution time significantly compared to the single-thread application. The execution time is faster up to 35%.

6.2.1 Cache-Miss Percentage Analysis

To further verify the performance improvement by data sharing in a shared cache memory, we examine the cache-miss percentage of each application. If the cache-miss percentage is reduced, the applications could speed up the execution time because the required data is ready in caches for processing and that reduces the extra time to load requested data from external memory. Cache misses, cache references are collected by using perf tool during the execution of an application. The cache-miss percentage is the percentage calculated from the cache misses over total cache references.

Figure 6.9 shows that cache-miss percentage is reduced significantly when running parallel-thread applications compared to single-thread applications. That explains why the execution time of parallel-thread applications is speeded up and it helps to improve the application's performance.

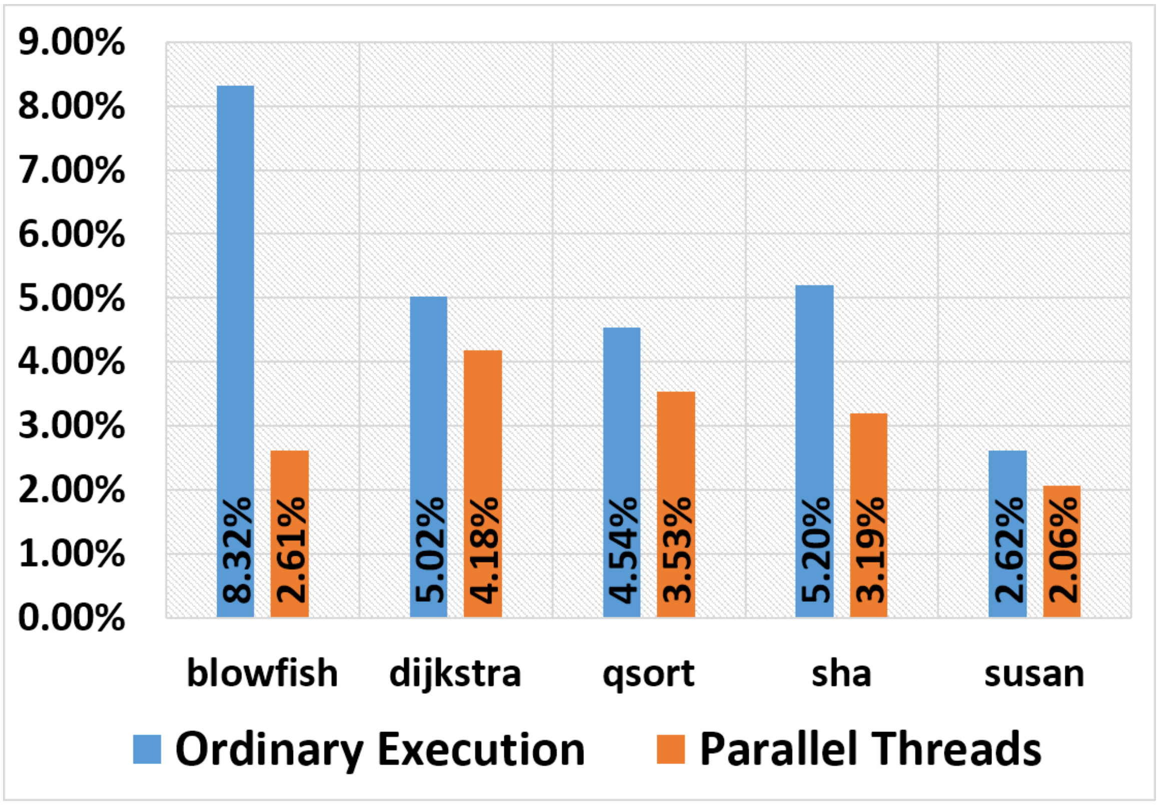

Figure 6.9: Cache-miss percentage reduction when threads run in two cores of Intel NUC

6.2.2 Results Analysis on Intel NUC

Experimental results show that execution time and cache-miss percentage are reduced significantly with parallel-thread applications compared to single-thread applications. That could be explained as follows:

- The replica thread (a duplicated version of main thread) helps to prefetch requested data into shared cache memory. Therefore, the requested data by the main thread is mostly available in the L2 cache. It reduces the time for the main thread to load data from external memory into caches memory.

- Intel NUC uses an inclusive cache hierarchy which helps for data sharing among processor cores. The inclusive cache hierarchy ensures that every block in L1 has a backup copy in L2. The advantage of an inclusive cache in multicore systems is the ease of locating a data block upon an L1 miss. On an L1 miss, either a copy of the requested block will be found in L2, or L2 will point to a modified version of the block in some L1, or the L2 miss will indicate that the request can be sent directly to the next level of the memory hierarchy. The inclusive cache does not need to implement the cache snooping-based protocol as in exclusive cache which is complex and consumes more time to look-up for the requested data in entire cache space.

- Homogeneous architecture does not need to implement the dedicated hardware such as Cache Coherent Interconnect (CCI) because all processor cores are identical and they run at the same clock frequency. The CCI block would introduce extra overhead by synchronizing data between heterogeneous cores to ensure the cache coherency.

Chapter 7

Conclusions and Future Works

This chapter concludes the advantages and drawbacks of existing memory subsystem architectures in terms of their suitability for use in 3D-MPSoCs. Existing multicore systems are heterogeneous architecture or homogeneous architecture. Heterogeneous architecture usually uses dedicated hardware such as Cache Coherent Interconnect (CCI) to ensure the cache coherency between different heterogeneous cores. On the other hand, homogeneous architecture deploys cache architecture with multiple layers (such as two or three layers of caches), but it does not need to use CCI.

Recent research studies in 3D-MPSoCs [9, 18] propose heterogeneous architecture as a novel architecture in 3D-MPSoCs to improve performance and save energy in multicore embedded systems. The high-performance tasks could be assigned to 'big' cores and the low-performance tasks could be assigned to 'LITTLE' cores to achieve maximum energy efficiency. However, our experimental works show that CCI hardware in heterogeneous architecture could introduce a substantial overhead/delay to multi-thread applications when the threads are allocated on different clusters of the processor cores.

On the other hand, the memory subsystem may utilize different hierarchical orders such as inclusive or exclusive. The different cache hierarchical orders have different impacts on the performance of data sharing in multi-threading and multicore processing. Data sharing is an important feature of 3D-MPSoCs that needs to understand thoroughly to overcome the performance hurdles of multi-threading in 3D-MPSoCs.

The next section summarizes our findings, and the last section presents a novel idea for future work in heterogeneous 3D-MPSoCs.

7.1 Conclusions

- Heterogeneous architectures bring some advantages to maximize the energy efficiency of 3D-MPSoCs but it should be used with caution. Multi-thread applications should be run on the same cluster of processor cores and should not be run on different clusters of cores to avoid significant overhead/delay caused by the Cache Coherent Interconnect. The overhead/delay could impact heavily on the overall performance of a multi-thread application.

- Exclusive cache hierarchy is preferred in traditional embedded systems due to the limit in cache capacity of conventional embedded processors. However, in modern multicore embedded systems and 3D-MPSoCs, exclusive cache shows the performance hurdles in data sharing of multi-threads in the shared cache memory. Upon a cache miss in L1, a processor core spends more efforts to find the requested data in all cache space.

- Inclusive cache hierarchy shows the performance improved substantially in multi-threading and multicore system by enabling data sharing among processor cores. Therefore, inclusive cache hierarchy helps to speed up the overall application's performance significantly.

7.2 Future Works

A new scheduling algorithm on heterogeneous 3D-MPSoCs could be developed based on the cache performance monitoring techniques to improve the performance of heterogeneous 3D-MPSoCs. To predict the temperature variations of an individual processor core, a programmer could use performance monitoring counters (PMC) hardware to observe the cache misses per each cycle. The cache-miss rate is a good indication signal for the look-ahead algorithm to control the core's temperature in advance. Appendix B presents a novel idea for scheduling algorithm in heterogeneous 3D-MPSoCs for future works.

Appendix A

Experimental Steps

This Appendix presents all experimental steps on Odroid-XU4 and Intel NUC.
Appendix A.1 presents the experimental steps on two 'LITTLE' cores on Odroid-XU4.
Appendix A.2 presents the experimental steps on two 'big' cores on Odroid-XU4.
Appendix A.3 presents the experimental steps on a pair of 'big-LITTLE' cores on Odroid-XU4.
Appendix A.4 presents the experimental steps on two homogeneous cores of Intel NUC.

A.1 Experimental steps on two LITTLE cores of Odroid-XU4

```
root@odroid:/home/odroid# cset shield -c 2,3
cset: --> activating shielding:
cset: moving 112 tasks from root into system cpuset...
[==================================================]%
cset: "system" cpuset of CPUSPEC(0-1,4-7) with 112 tasks running
cset: "user" cpuset of CPUSPEC(2-3) with 0 tasks running

#1. blowfish
odroid@odroid:~/MibenchModified/blowfish_modified$ time cset shield --
    exec ./runme_large.sh
cset: **> 1 tasks are not movable, impossible to move
cset: --> last message, executed args into cpuset "/user", new pid is:
    2029
```

```
real	0m55.093s
user	0m54.530s
sys	0m0.540s

odroid@odroid:~/MibenchModified/blowfish_modified$ time cset shield --
    exec ./parallel_command.py
cset: **> 1 tasks are not movable, impossible to move
cset: --> last message, executed args into cpuset "/user", new pid is:
    2037
real	1m2.326s
user	2m3.085s
sys	0m1.320s

#2. dijkstra
odroid@odroid:~/MibenchModified/dijkstra_modified$ time cset shield --
    exec ./runme_large.sh
cset: **> 1 tasks are not movable, impossible to move
cset: --> last message, executed args into cpuset "/user", new pid is:
    1804
real	0m42.542s
user	0m34.955s
sys	0m7.580s

odroid@odroid:~/MibenchModified/dijkstra_modified$ time cset shield --
    exec ./parallel_command.py
cset: **> 1 tasks are not movable, impossible to move
cset: --> last message, executed args into cpuset "/user", new pid is:
    1810
real	0m48.468s
user	1m17.685s
sys	0m17.995s

#3. qsort
odroid@odroid:~/MibenchModified/qsort_modified$ time cset shield --exec
    ./runme_large.sh
cset: **> 1 tasks are not movable, impossible to move
cset: --> last message, executed args into cpuset "/user", new pid is:
    2086
```

```
real    0m45.352s
user    0m27.845s
sys     0m17.325s

odroid@odroid:~/MibenchModified/qsort_modified$ time cset shield --exec
    ./parallel_command.py
cset: **> 1 tasks are not movable, impossible to move
cset: --> last message, executed args into cpuset "/user", new pid is:
    2093

real    0m50.562s
user    1m1.055s
sys     0m39.815s

#4. sha
odroid@odroid:~/MibenchModified/sha_modified$ time cset shield --exec ./
    runme_large.sh
cset: **> 1 tasks are not movable, impossible to move
cset: --> last message, executed args into cpuset "/user", new pid is:
    2173
real    0m54.519s
user    0m29.970s
sys     0m24.525s

odroid@odroid:~/MibenchModified/sha_modified$ time cset shield --exec ./
    parallel_command.py
cset: **> 1 tasks are not movable, impossible to move
cset: --> last message, executed args into cpuset "/user", new pid is:
    2182
real    1m5.822s
user    1m9.940s
sys     1m1.510s

#5. susan
odroid@odroid:~/MibenchModified/susan_modified$ time cset shield --exec
    ./runme_large.sh
cset: **> 1 tasks are not movable, impossible to move
cset: --> last message, executed args into cpuset "/user", new pid is:
    2246
```

```
real    0m54.011s
user    0m32.260s
sys     0m21.725s

odroid@odroid:~/MibenchModified/susan_modified$ time cset shield --exec
    ./parallel_command.py
cset: **> 1 tasks are not movable, impossible to move
cset: --> last message, executed args into cpuset "/user", new pid is:
    2254
real    1m2.069s
user    1m13.160s
sys     0m50.825s
```

Listing A.1: Experiment steps on two LITTLE cores

A.2 Experimental steps on two big cores of Odroid-XU4

```
root@odroid:/home/odroid# cset shield -c 4,5
cset: --> activating shielding:
cset: moving 111 tasks from root into system cpuset...
[==================================================]%
cset: "system" cpuset of CPUSPEC(0-3,6-7) with 111 tasks running
cset: "user" cpuset of CPUSPEC(4-5) with 0 tasks running

#1. blowfish
root@odroid:/home/odroid/MibenchModified/blowfish_modified# time cset
    shield --exec ./runme_large.sh
cset: --> last message, executed args into cpuset "/user", new pid is:
    1361

real    0m55.686s
user    0m54.270s
sys     0m0.495s
root@odroid:/home/odroid/MibenchModified/blowfish_modified# time cset
    shield --exec ./parallel_command.py
```

```
cset: --> last message, executed args into cpuset "/user", new pid is:
    1374

real    1m2.232s
user    2m2.385s
sys     0m1.180s

#2. dijkstra
root@odroid:/home/odroid/MibenchModified/dijkstra_modified# time cset
    shield --exec ./runme_large.sh
cset: --> last message, executed args into cpuset "/user", new pid is:
    1390

real    0m42.958s
user    0m35.040s
sys     0m7.905s
root@odroid:/home/odroid/MibenchModified/dijkstra_modified# time cset
    shield --exec ./parallel_command.py
cset: --> last message, executed args into cpuset "/user", new pid is:
    1392

real    0m47.426s
user    1m15.730s
sys     0m17.965s

#3. qsort
root@odroid:/home/odroid/MibenchModified/qsort_modified# time cset
    shield --exec ./runme_large.sh
cset: --> last message, executed args into cpuset "/user", new pid is:
    1400

real    0m45.145s
user    0m28.160s
sys     0m16.975s
root@odroid:/home/odroid/MibenchModified/qsort_modified# time cset
    shield --exec ./parallel_command.py
cset: --> last message, executed args into cpuset "/user", new pid is:
    1402
```

```
real    0m50.110s
user    0m59.350s
sys     0m40.720s

#4. sha
root@odroid:/home/odroid/MibenchModified/sha_modified# time cset shield
    --exec ./runme_large.sh
cset: --> last message, executed args into cpuset "/user", new pid is:
    1415

real    0m58.488s
user    0m31.430s
sys     0m27.055s
root@odroid:/home/odroid/MibenchModified/sha_modified# time cset shield
    --exec ./parallel_command.py
cset: --> last message, executed args into cpuset "/user", new pid is:
    1429

real    1m4.423s
user    1m8.575s
sys     1m0.130s

#5. susan
root@odroid:/home/odroid/MibenchModified/susan_modified# time cset
    shield --exec ./runme_large.sh
cset: --> last message, executed args into cpuset "/user", new pid is:
    1438

real    0m55.419s
user    0m32.315s
sys     0m23.090s
root@odroid:/home/odroid/MibenchModified/susan_modified# time cset
    shield --exec ./parallel_command.py
cset: --> last message, executed args into cpuset "/user", new pid is:
    1444

real    1m1.249s
user    1m13.260s
sys     0m48.995s
```

Listing A.2: Experimental steps on two big cores

A.3 Experimental steps on a pair of big-LITTLE cores of Odroid-XU4

```
root@odroid:/home/odroid# cset shield -c 3,4
cset: --> activating shielding:
cset: moving 109 tasks from root into system cpuset...
[==================================================]%
cset: "system" cpuset of CPUSPEC(0-2,5-7) with 109 tasks running
cset: "user" cpuset of CPUSPEC(3-4) with 0 tasks running

#1. blowfish
root@odroid:/home/odroid/MibenchModified/blowfish_modified# time cset
    shield --exec ./runme_large.sh
cset: --> last message, executed args into cpuset "/user", new pid is:
    1358

real    0m56.558s
user    0m54.750s
sys     0m0.800s
root@odroid:/home/odroid/MibenchModified/blowfish_modified# time cset
    shield --exec ./parallel_command.py
cset: --> last message, executed args into cpuset "/user", new pid is:
    1367

real    0m56.715s
user    1m49.485s
sys     0m1.140s

#2. dijkstra
root@odroid:/home/odroid/MibenchModified/dijkstra_modified# time cset
    shield --exec ./runme_large.sh
cset: --> last message, executed args into cpuset "/user", new pid is:
    1387
```

```
real    1m4.363s
user    0m55.600s
sys     0m8.750s
root@odroid:/home/odroid/MibenchModified/dijkstra_modified# time cset
    shield --exec ./parallel_command.py
cset: --> last message, executed args into cpuset "/user", new pid is:
    1389

real    1m24.319s
user    1m9.090s
sys     0m15.185s

#3. qsort
root@odroid:/home/odroid/MibenchModified/qsort_modified# time cset
    shield --exec ./runme_large.sh
cset: --> last message, executed args into cpuset "/user", new pid is:
    1406

real    1m5.751s
user    0m45.360s
sys     0m20.375s
root@odroid:/home/odroid/MibenchModified/qsort_modified# time cset
    shield --exec ./parallel_command.py
cset: --> last message, executed args into cpuset "/user", new pid is:
    1409

real    1m27.414s
user    0m53.590s
sys     0m33.780s

#4. sha
root@odroid:/home/odroid/MibenchModified/sha_modified# time cset shield
    --exec ./runme_large.sh
cset: --> last message, executed args into cpuset "/user", new pid is:
    1424

real    1m21.766s
user    0m49.605s
sys     0m32.155s
```

```
root@odroid:/home/odroid/MibenchModified/sha_modified# time cset shield
    --exec ./parallel_command.py
cset: --> last message, executed args into cpuset "/user", new pid is:
    1428

real    1m53.475s
user    1m1.575s
sys     0m51.850s

#5. susan
root@odroid:/home/odroid/MibenchModified/susan_modified# time cset
    shield --exec ./runme_large.sh
cset: --> last message, executed args into cpuset "/user", new pid is:
    1450

real    1m24.628s
user    0m54.160s
sys     0m30.455s
root@odroid:/home/odroid/MibenchModified/susan_modified# time cset
    shield --exec ./parallel_command.py
cset: --> last message, executed args into cpuset "/user", new pid is:
    1455

real    1m47.474s
user    1m3.765s
sys     0m43.535s
```

Listing A.3: Experimental steps on a pair of big-LITTLE cores

A.4 Experimental steps on Intel NUC

```
## Creating a shield ##

root@ntumsh-NUC5PGYH:/home/dangadm/MibenchModified/# cset shield -c 2,3
cset: --> activating shielding:
cset: moving 344 tasks from root into system cpuset...
[==================================================]%
cset: "system" cpuset of CPUSPEC(0-1) with 344 tasks running
```

```
cset: "user" cpuset of CPUSPEC(2-3) with 0 tasks running

#1 Blowfish

blowfish_modified# perf stat -e cache-references,cache-misses cset
   shield --exec ./parallel_command.py
cset: --> last message, executed args into cpuset "/user", new pid is:
   8636

 Performance counter stats for 'cset shield --exec ./parallel_command.py
   ':

     1,133,459,212      cache-references
        88,888,979      cache-misses              #    7.842 % of all
   cache refs

     109.048398330 seconds time elapsed

blowfish_modified# perf stat -e cache-references,cache-misses cset
   shield --exec ./runme_large.sh
cset: --> last message, executed args into cpuset "/user", new pid is:
   8657

 Performance counter stats for 'cset shield --exec ./runme_large.sh':

       629,950,079      cache-references
       157,142,428      cache-misses              #   24.945 % of all
   cache refs

     172.850781659 seconds time elapsed

#2 Dijkstra

dijkstra_modified# perf stat -e cache-references,cache-misses cset
   shield --exec ./parallel_command.py
cset: --> last message, executed args into cpuset "/user", new pid is:
   8584
 Performance counter stats for 'cset shield --exec ./parallel_command.py
    ':

    18,921,929,327      cache-references
       779,203,669      cache-misses              #    4.118 % of all
    cache refs

     263.342451404 seconds time elapsed
dijkstra_modified# perf stat -e cache-references,cache-misses cset
    shield --exec ./runme_large.sh
cset: --> last message, executed args into cpuset "/user", new pid is:
    8549

 Performance counter stats for 'cset shield --exec ./runme_large.sh':

    10,369,663,514      cache-references
       520,790,448      cache-misses              #    5.022 % of all
    cache refs

     330.717180611 seconds time elapsed


#3 Qsort

qsort_modified# perf stat -e cache-references,cache-misses cset shield
    --exec ./parallel_command.py
cset: --> last message, executed args into cpuset "/user", new pid is:
    9330

 Performance counter stats for 'cset shield --exec ./parallel_command.py
    ':

    32,259,173,758      cache-references
     1,139,820,819      cache-misses              #    3.533 % of all
    cache refs

     526.764664465 seconds time elapsed
```

```
qsort_modified# perf stat -e cache-references,cache-misses cset shield
    --exec ./runme_large.sh
cset: --> last message, executed args into cpuset "/user", new pid is:
    9400

 Performance counter stats for 'cset shield --exec ./runme_large.sh':

    18,377,453,194      cache-references
       834,863,756      cache-misses              #    4.542 % of all
    cache refs

     661.465216584 seconds time elapsed


#4 Sha

sha_modified# perf stat -e cache-references,cache-misses cset shield --
    exec ./parallel_command.py
cset: --> last message, executed args into cpuset "/user", new pid is:
    21370

 Performance counter stats for 'cset shield --exec ./parallel_command.py
    ':

     7,970,190,481      cache-references
       253,976,801      cache-misses              #    3.193 % of all
    cache refs

     126.120263065 seconds time elapsed

sha_modified# perf stat -e cache-references,cache-misses cset shield --
    exec ./runme_large.sh
cset: --> last message, executed args into cpuset "/user", new pid is:
    21401

 Performance counter stats for 'cset shield --exec ./runme_large.sh':

     4,615,553,263      cache-references
```

```
        239,874,299        cache-misses                #    5.20% % of all
    cache refs

      180.422914162 seconds time elapsed



#5 Susan

susan_modified#  perf stat -e cache-references,cache-misses cset shield
    --exec ./parallel_command.py
cset: --> last message, executed args into cpuset "/user", new pid is:
    23933

 Performance counter stats for 'cset shield --exec ./parallel_command.py
    ':

     11,868,888,316        cache-references
        244,257,113        cache-misses                #    2.058 % of all
    cache refs

      166.997445179 seconds time elapsed



susan_modified#  perf stat -e cache-references,cache-misses cset shield
    --exec ./runme_large.sh
cset: --> last message, executed args into cpuset "/user", new pid is:
    24168

 Performance counter stats for 'cset shield --exec ./runme_large.sh':

      6,393,938,329        cache-references
        167,594,296        cache-misses                #    2.621 % of all
    cache refs

      184.353097592 seconds time elapsed
```

Listing A.4: Experimental steps on Intel NUC

Appendix B

A Novel Thermal-Aware Scheduling Algorithm for Heterogeneous 3D-MPSoCs

B.1 Sorting cores' temperature by increasing order

Figure B.1: Sorting cores temperature by increasing order.

We sort cores' temperature by increasing order with the assumption that, each core has a separate thermal sensor. Currently Intel and AMD processor already have thermal sensors in each processor core. We did not see any ARM or PowerPC processors have separate thermal sensor in each core yet.
Nevertheless, we still can use L1 cache misses rate per instruction cycle for the energy consumption estimation.

B.2 Swapping processes between pair of cores

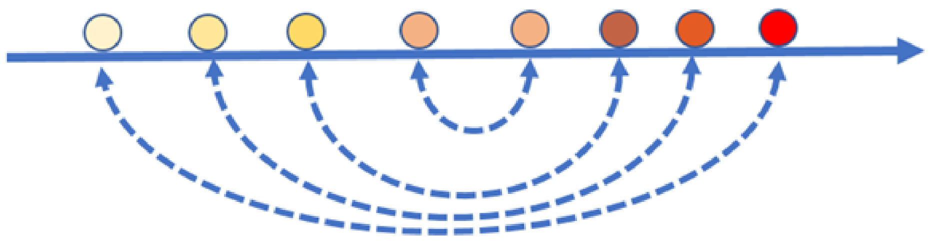

Figure B.2: Swap processes between pair of cores.

We swap processes between the hottest core with the coolest core, the second hottest core with the second coolest core, and so on . . .

B.3 After T interval time, repeat step 1 and step 2

T interval can be dynamically calculated based on:

- Based on the number of cache misses in L1 cache level over a time period.
- The thermal gap between the hottest core and coolest core.
- We should investigate for an algorithm that dynamically calculate this T interval to be an optimal value.
- If the T is too small that mean we do the swapping too often, it introduces a lot of overhead. But if T is too big, that mean we could not cool down the processor efficiently in a shortest time.

Bibliography

[1] Wei Zang and Ann Gordon-Ross. A survey on cache tuning from a power/energy perspective. *ACM Comput. Surv.*, 45(3):32:1–32:49, July 2013.

[2] Chuanjun Zhang, Frank Vahid, and Walid Najjar. A highly configurable cache for low energy embedded systems. *ACM Trans. Embed. Comput. Syst.*, 4(2):363–387, May 2005.

[3] M. A. Awan and S. M. Petters. Enhanced race-to-halt: A leakage-aware energy management approach for dynamic priority systems. In *2011 23rd Euromicro Conference on Real-Time Systems*, pages 92–101, July 2011.

[4] S. Chakraborty, S. Das, and H. K. Kapoor. Power aware cache miss reduction by energy efficient victim retention. In *2015 19th International Symposium on VLSI Design and Test*, pages 1–6, June 2015.

[5] X. Zhou, Y. Xu, Y. Du, Y. Zhang, and J. Yang. Thermal management for 3d processors via task scheduling. In *2008 37th International Conference on Parallel Processing*, pages 115–122, Sept 2008.

[6] X. Zhou, J. Yang, Y. Xu, Y. Zhang, and J. Zhao. Thermal-aware task scheduling for 3d multicore processors. *IEEE Transactions on Parallel and Distributed Systems*, 21(1):60–71, Jan 2010.

[7] V. Chaturvedi, A. K. Singh, W. Zhang, and T. Srikanthan. Thermal-aware task scheduling for peak temperature minimization under periodic constraint for 3d-mpsocs. In *2014 25nd IEEE International Symposium on Rapid System Prototyping*, pages 107–113, Oct 2014.

[8] J. Li, M. Qiu, J. Niu, T. Chen, and Y. Zhu. Real-time constrained task scheduling in 3d chip multiprocessor to reduce peak temperature. In *2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, pages 170–176, Dec 2010.

[9] T. H. Tsai, Y. S. Chen, X. X. He, and C. Y. Li. Stem: A thermal-constrained real-time scheduling for 3d heterogeneous-isa multicore processors. *IEEE Transactions on Computers*, 67(6):874–889, June 2018.

[10] C. H. Liao, C. H. P. Wen, and K. Chakrabarty. An online thermal-constrained task scheduler for 3d multi-core processors. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 351–356, March 2015.

[11] Chien-Hui Liao and Hung-Pin Wen. Performance validation of dynamic-remapping-based task scheduling on 3d multi-core processors. In *Proceedings of Technical Program of 2012 VLSI Design, Automation and Test*, pages 1–4, April 2012.

[12] M. Cox, A. K. Singh, A. Kumar, and H. Corporaal. Thermal-aware mapping of streaming applications on 3d multi-processor systems. In *The 11th IEEE Symposium on Embedded Systems for Real-time Multimedia*, pages 11–20, Oct 2013.

[13] Z. Gan, M. Zhang, Z. Gu, and J. Zhang. Minimizing energy consumption for embedded multicore systems using cache configuration and task mapping. In *2016 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pages 328–334, Oct 2016.

[14] Antonio Artes, Jose L. Ayala, Jos Huisken, and Francky Catthoor. Survey of low-energy techniques for instruction memory organisations in embedded systems. *J. Signal Process. Syst.*, 70(1):1–19, January 2013.

[15] Odroid-XU4 odroid-xu4, the heterogeneous multi-processing (hmp) solution. http://www.hardkernel.com/main/products/prdt_info.php.

[16] Intel-NUC intel nuc kit nuc5ppyh. https://www.intel.sg/content/dam/www/public/us/en/documents/product-briefs/nuc-pentium-brief.pdf.

[17] ARM Limited. big.LITTLE Technology: The Future of Mobile. https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf, 2013.

[18] B. N. Swamy, A. Ketterlin, and A. Seznec. Hardware/software helper thread prefetching on heterogeneous many cores. In *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, pages 214–221, Oct 2014.