HALMSTAD UNIVERSITY

Master's Programme in Embedded and Intelligent Systems, 120 credits

MASTER THESIS

Acceleration of Parallel Applications by Moving Code Instead of Data

Farzad Farahaninia

Computer science and engineering, 30 credits

Halmstad 2014

Acceleration of Parallel Applications by Moving Code Instead of Data

School of Information Technology
Halmstad University
in collaboration with
Ericsson AB

Author: Farzad Farahaninia

Supervisors: Tomas Nordstrom
Halmstad University

Martin Lundqvist
Ericsson AB

Examiner: Tony Larsson

Halmstad, 2014

Acknowledgments

I would like to express my deepest gratitude to Martin Lundqvist, my supervisor at Ericsson, for his encouragement, motivational support and persistent help. I would also like to extend my appreciation to David Engdal, Peter Brauer and the rest of the baseband research team at Ericsson, for providing me a very motivational research environment.

I cannot find words to express my gratitude to Professor Tomas Nordstrom, my supervisor at Halmstad University, for the patient guidance, support and encouragement.

Special thanks to Sebastian Raase, my friend and mentor. Your insights have always been inspirational and motivating, and your friendship is invaluable.

Last but most importantly, I would like to thank my Family for providing me with the opportunity to pursue my dreams. This work is dedicated to them with respect and gratitude.

Abstract

After the performance improvement rate in single-core processors decreased in 2000s, most CPU manufacturers have steered towards parallel computing. Parallel computing has been in the spotlight for a while now. Several hardware and software innovations are being examined and developed, in order to improve the efficiency of parallel computing.

Signal processing is an important application area of parallel computing, and this makes parallel computing interesting for Ericsson AB, a company that among other business areas, is mainly focusing on communication technologies. The Ericsson baseband research team at Lindholmen, has been developing a small, experimental basic operating system (BOS) for research purposes within the area of parallel computing.

One major overhead in parallel applications, which increases the latency in applications, is the communication overhead between the cores. It had been observed that in some signal processing applications, it is common for some tasks of the parallel application to have a large data size but a small code size. The question was risen then, could it be beneficial to move code instead of data in such cases, to reduce the communication overhead.

In this thesis work the gain and practical difficulties of moving code are investigated through implementation. A method has been successfully developed and integrated into BOS to move the code between the cores on a multi-core architecture. While it can be a very specific class of applications in which it is useful to move code, it is shown that it is possible to move the code between the cores with zero extra overhead.

Contents

1. Introduction **3**
- 1.1. Moving code instead of data . . . 3
- 1.2. Research question . . . 6
- 1.3. Related work . . . 7
  - 1.3.1. Overlays . . . 7
  - 1.3.2. Dynamic code deployment using ActiveSpaces . . . 8
  - 1.3.3. In Situ Visualization for Large-Scale Simulations . . . 9

2. Methodology **11**
- 2.1. Research method . . . 11
- 2.2. Tools . . . 11
  - 2.2.1. Hardware architecture . . . 11
  - 2.2.2. Operating system . . . 12
- 2.3. Benchmarking . . . 12

3. Background **15**
- 3.1. Parallel Architectures . . . 15
- 3.2. Adapteva's Epiphany architecture . . . 16
- 3.3. Parallel programming paradigms . . . 17
- 3.4. BOS . . . 19
  - 3.4.1. Graph description . . . 23
  - 3.4.2. BOS Operation . . . 26
  - 3.4.3. Internal representation of graphs . . . 27
  - 3.4.4. Core states . . . 30
  - 3.4.5. Trace functionality . . . 31
- 3.5. Compilers . . . 33
- 3.6. The linker . . . 34
- 3.7. Dynamic linking . . . 36

4. Evaluating code movement vs data movement **37**
- 4.1. Storage cores, why are they needed? . . . 38
- 4.2. Changes to graph representation . . . 40
- 4.3. Changes to BOS operation . . . 41
- 4.4. Manipulating the memory layout . . . 43
- 4.5. Further optimization . . . 49
- 4.6. Benchmarking . . . 49
- 4.7. Results and analysis . . . 51

5. Conclusions and future work **57**
5.1. Future work 57

A. Linker scripts **59**
A.1. Accessing linker defined symbols from code 62

B. Implementation **65**
B.1. Memory layout of worker cores 65
B.2. Memory Layout of Storage Cores 67

Bibliography **71**

1

Introduction

It is no longer possible to keep improving the performance of single core processors at a rate as high as before. In order to maintain a high growth rate in computational performance, using parallel computing is essential. Several hardware and software innovations are being examined and developed, in order to improve the efficiency of parallel computing.

Signal processing is an important application area of parallel computing, and this makes parallel computing interesting for Ericsson AB, a company that among other business areas, is mainly focusing on communication technologies. The Ericsson baseband research team at Lindholmen, has been developing a small, experimental basic operating system (BOS) for research purposes within the area of parallel computing (section 3.4). BOS is designed to operate in a very dynamic environment, and it provides very good tracing functionalities, which makes it a great research tool.

One major overhead in parallel applications, which increases the latency in applications, is the communication overhead between the cores. It had been observed that in some signal processing applications, it is common for some tasks of the parallel application to have a large data size but a small code size. The interesting question was risen then, could it be beneficial to move code instead of data in such cases, to reduce the communication overhead.

In this section, we will discuss how moving code can reduce the communication overhead, and consequently, the total latency of a parallel application. Furthermore, we will review some of the research efforts related to this thesis work.

1.1. Moving code instead of data

In order to present the research topic in this thesis work, we will briefly review the hardware and software domain in which this research is conducted.

In modern computers, parallelism is deployed in many different forms including: vector instructions, multi-threaded cores, many-core processors, multiple processors, graphics engines, and parallel co-processors [10]. This thesis work is is conducted on a many-core hardware architecture, Adapteva's Epiphany, which is a MIMD (multiple instruction multiple data) architecture.

In order to use a parallel hardware, a parallel application needs to be designed. The main idea behind parallel applications is splitting the work into smaller tasks. Each task will be responsible for carrying out part of the computation. The tasks will then be mapped to different cores of the hardware to be run concurrently to achieve speedup.

Parallel applications can often be described using a directed graph, like the one in figure 1.1.a. In such a graph, each node will correspond to a task, and edges can symbolize either precedence constraints (in a control flow description) or data transfer (in a data flow application). In a control flow application description, the edges describe the order of execution, whereas, in a data flow application they describe the flow of data: where the data comes from and where is it destined for. In this thesis work we will focus on data flow application description and assume that all edges indicate the path and the direction over which the data has to be transferred.

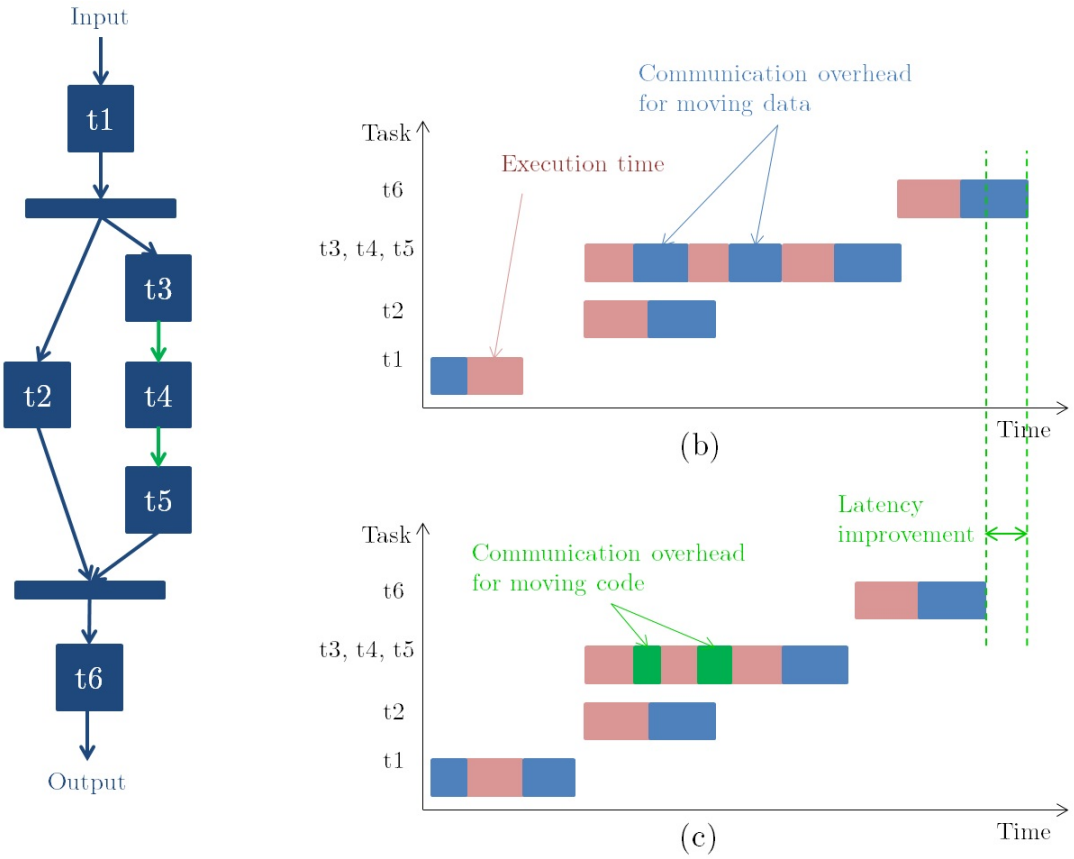


Figure 1.1.: Comparison of code movement versus data movement. (a) shows a data-flow parallel application in which every task is mapped to a different core. (b) shows the occurance of the events in a time line, if all tasks are executed in a sequence and data is moved between the cores. (c) shows the events for the same application, but this time instead of moving data between tasks t3-t4 and t4-t5, the task code is delivered to the core. If delivering the code takes shorter time and the possible overheads are small enough, speed up in the total latency will be achieved.

In the application described in figure 1.1.a, each node represents a task, and we will assume that each task is mapped to a different core. The input data will enter the graph from the *Input* channel. Each task's processed data will be copied to the core running the next task, and finally, the processed data will exit through the *Output* channel. Details of input and output channels are not of our concern.

Figure 1.1.b shows the events during execution of this application in a time line: pink bars represent the execution time of tasks, and the blue bars represent the communication overhead of copying the output data to the next task (core). As can be seen, consecutive tasks can start only when the previous task(s) had finished execution and its output data has been copied out to the core running the next task.

To measure the performance of parallel applications, two of the commonly used criteria

include *latency* and *throughput*. Latency is the elapsed time between the submission of data to the system until it has been processed. Throughput is the rate at which the system outputs data. To show the possible gain of moving code, we will use latency as criteria. In the time line in figure 1.1.b, the latency is the total time measured from the moment the input data enters the system, until the last task is finished and the output data is copied out.

In the case that the code size for tasks t4 and t5 are smaller than the data size that needs to be delivered to them, the possible improvement that we can introduce to this application is that instead of moving a big amount of data between the cores running these tasks, we deliver the code for these tasks to the same core that has executed the task t3. That is, the core that executed the task t3, will retain the output data and the code for the tasks t4 and t5 will be delivered to it. Moving code can of course introduce new overheads (run time linking for example), but if these overheads can be reduced or eliminated, and the code size is small enough, we can possibly reduce the total latency of the application by using this method. This is shown in figure 1.1.c: the green bars represent the communication time for moving code, and as can be seen, this communication overhead is smaller than that of moving data for the respective tasks. Since the next task can start immediately after delivery of the code (given that the extra overheads of moving code are small or non existent), the total latency of the application can be reduced.

1.2. Research question

While designing Parallel applications, it has been observed that in some cases (specially in signal analysis applications) it can be the case to have a rather large data size which needs to be passed between the tasks, and a fairly small code size for those certain tasks. The larger the data size, the longer the communication overhead, as well as more load on the communication network of the hardware. An interesting question then rises: In case of smaller code size, can it be beneficial to move the code instead of data to reduce the communication overhead?

This can cause a significant improvement in the performance of parallel applications on many-core data-flow applications, since communication overhead plays an important role in the total latency of the application; however, some technical difficulties need to be taken under consideration. For example, moving the code to the hardware cores can raise the need of dynamic linking. Performing dynamic linking by itself will introduce a new overhead, which might outweigh the achieved improvement. Supporting such functionality can also use up the available memory of a core, which, in case of Epiphany is very scarce. All these reasons make dynamic linking less attractive as a solution. In this work, a solution was provided which avoids dynamic linking during run-time. This, among other considerations and challenges will be elaborated in section 4.

The research question whose validity is investigated in this thesis work is: *Despite the presence of some extra overheads, can it be beneficial to move the code instead of data, in case the code size is smaller?*. Furthermore, we will try to address the following:

1. What are the challenges of moving and executing code on a different core on a parallel hardware, and how can they be tackled?
2. Does doing so introduce new overheads?
3. What is the size threshold at which it becomes beneficial to move the code instead of the data?

Parallel applications that we have focused on are made of tasks and these tasks can either come in a sequence or in a fork-join pattern (where the tasks divide into multiple execution branches). It should be mentioned that while it is possible to move the code between the tasks where there is a fork-join in the graph, in this thesis work we have focused on moving code between the tasks that come in sequence. This is due to the fact that moving code between the tasks which come after a fork-join can introduce too much complications for too little gain, while focusing on sequential tasks would be enough for our research purposes as this work is mainly a proof of concept.

1.3. Related work

To the best of author's knowledge, this is the first attempt to move code between the cores on a MIMD many-core hardware in order to improve the latency of parallel applications; however, the techniques used in this work for moving code are quite old, and the idea of moving code is has also been investigated on different kind of hardware and a different scale. This section summarizes previous research efforts related to this thesis work. The related work can be divided into two main categories:

1. Delivering parts of code or data to internal memory during run-time. This technique is called *overlay* and is introduced in 1.3.1.
2. Attempts to reduce communication overhead for peta-scale data sizes on super computers, by performing more post-processing computations in situ and eliminating the need for data transfers. Examples of this are briefly reviewed in 1.3.3 and 1.3.2.

1.3.1. Overlays

Overlay is the process of transferring a block of program code or other data into internal memory to replace what is already stored. This technique dates back to before 1960 and is still in use in environments with memory constraints. In this technique, the code is divided into a tree of segments (figure 1.2) called "overlay segments". Sibling segments in the overlay tree share the same memory. In this example, segments A and

D share the same meory, B and C share the same memory, and E and F share the same memory as well. The object files or individual object code segments are assigned to overlay segments by the programmer. When the program starts, the root segment is loaded, which is the entry point of the program. Whenever a routine makes a downward intersegment call, the overlay manager ensures that the call target is loaded [8].

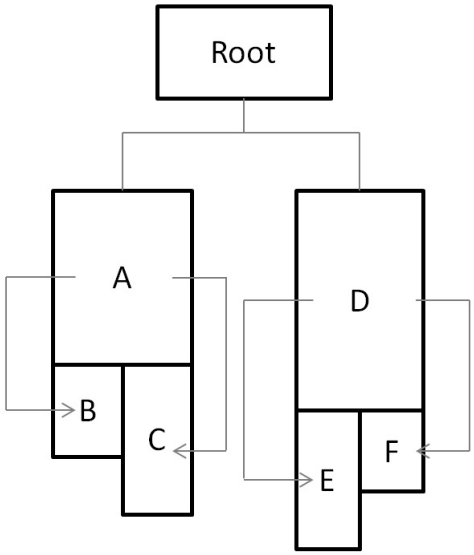


Figure 1.2.: Overlay tree

This technique is useful when there are memory constraints on the hardware, because it let's the programmer create a program which doesn't fit into the internal memory. Today this technique is in use in embedded processors which do not provide an MMU unit.

1.3.2. Dynamic code deployment using ActiveSpaces

Managing the large volumes of data in emerging scientific and engineering simulations is becoming more challenging, since the data has to be extracted off the computing nodes and delivered to consumer nodes. One approach trying to address this challenge is offloading expensive I/O operations to a smaller set of dedicated computing nodes known as *staging area*. However, using this approach the data still has to be moved from staging area to the consumer nodes. An alternative to this, is moving the data-processing code to the staging area using ActiveSpaces framework [5]. The ActiveSpaces framework provides (1) programming support for defining the data processing routines, called data kernels, to be executed on data objects on the staging area, and (2) run-time mechanisms for transporting the binary codes associated with these data kernels to the staging area and executing them in parallel on the staging nodes that host the specified data objects. This relates to our topic since it implements a way of transferring kernels during run-time to avoid moving huge chunks of data; however, this is implemented on a supercomputer operating Linux, and uses dynamic linking

functionalities of Linux. Dymamic linking was avoided in this thesis work, since it had to be implemented from scratch, the memory limitations and the fact that it could introduce new overheads which could counter balance the possible speed up.

1.3.3. In Situ Visualization for Large-Scale Simulations

Supercomputers have been a tool for simulations in many scientific areas for a long time. As machines have got more powerful, the size and complexity of problems has also grown. As a result, the size of the data these simulations generate has grown significantly and is now occupying petabytes. As the supercomputer time is expensive, the power of supercomputers is mainly used for simulations while the visualization and data analysis are offline tasks. At peta-scale, transferring raw simulation data to storage devices or visualization machines is cumbersome. An alternative approach is to reduce or transform data in situ on the same machine as the simulation is running to minimize the data or information requiring storage or transfer [14], [9]. In situ processing eliminates the need for transferring large amounts of raw data and thus is highly scalable. This is similar to this thesis work in the sense that it avoids data-movement and delivers the code to the nodes on a parallel computing platform which are producing simulation outputs; however, in these cases the data size is orders of magnitude larger than the code size, which is not the case in domain of this thesis work. Also, these experiences are only for visualization purposes and the used platform is a supercomputer.

2

Methodology

This section will describe the chosen research method to investigate the research question. Our main question is to find out whether or not it is beneficial to move the code instead of the data, and what are the difficulties of doing so.

2.1. Research method

In order to evaluate the value of code movement we will take an experimental approach and implement the concept on a real hardware. This has the great advantage that along the way we will have the chance to discover the technical challenges of moving code, which can be a very important factor for application designers.

In addition to the research question, we will also try to answer the following questions:

1. What are the challenges of moving and executing code on a different core on a parallel hardware, and how can they be tackled?
2. Does doing so introduce new overheads?
3. What is the size threshold at which it becomes beneficial to move the code instead of the data?

2.2. Tools

2.2.1. Hardware architecture

Several different parallel architectures exist which are being used in the industry and research. Some of the examples which were considered are Epiphany, KALRAY and Ambric. Among these, Epiphany was chosen because of availability, high performance and low power consumption which is suitable for many embedded applications that Ericsson is working on. The specific computer on which the experiments are carried out is Parallella-16 [2], a single-board computer based on Adapteva's 16 core Epiphany

chip (E16G301) [3]. A gcc based tool-chain is provided to create applications to run on this board.

2.2.2. Operating system

In order to run many tasks on a hardware, it is beneficial to take advantage of a basic operating system which can facilitate communication and synchronization between the tasks; therefore this work will extend an existing experimental basic operating system (BOS) under development at Ericsson. BOS facilitates inter-core communication, synchronization, mapping of computation support on different cores, and, description of parallel applications using different paradigms. It also provides accurate tracing functionalities for time measurement and benchmarking purposes. This makes it a great tool for research purposes. In this thesis work, moving code is added as a new feature to BOS, and its tracing functionality is used to benchmark the implemented method.

2.3. Benchmarking

To evaluate the implemented solution, two different graph configurations of an artificial application are created: one which takes advantage of the implemented solution and attempts to deliver code to the core executing tasks (figure 2.2), and one which executes each task on a different core and attempts to move the data between them (figure 2.1). The performance measurements of the two are then compared together (section 4.6 and 4.7). BOS provides a facility to measure the duration of different events during program execution(section 3.4.5).

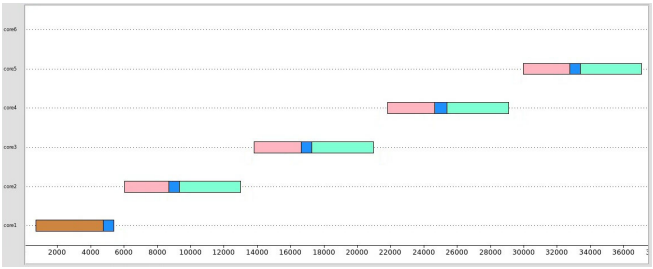


Figure 2.1.: Events occurring on different cores of the hardware while executing each task on a different core and moving data between the cores. The color pink is task execution, cyan is the time for moving data and dark blue is the platform overhead to find a core for the next task (explained in section3.4.

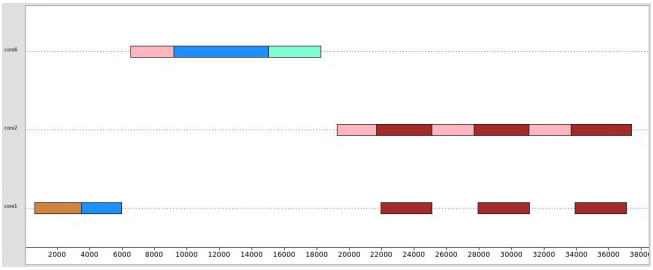


Figure 2.2.: Events on the hardware with core 2 executing tasks sequentially with different kernel code being delivered to it. Details of these applications are explained in sections 4.6 and 4.7.

Please note that when tasks are executed on the hardware, the execution time will be the same whether the code is moved or not. Our solution will not affect the execution time and it can only improve the communication overhead. Therefore the execution time is not of any interest to us and what we will try to compare in order to reach a conclusion is the communication time.

In the next chapter, we will review some of the essential background topics and we will also introduce BOS. The challenges and considerations of moving code and the suggested solution are discussed in chapter 4. In order to benchmark the implemented solution, an artificial application with two different configurations (section 4.6) is created to force data movement in one case and code movement in another case.

3

Background

Before we can perform our experiments and evaluation, we will need some background information on parallel architectures, the BOS operating system and compilers.

3.1. Parallel Architectures

In this section we will have a short review of the reasons why parallel computing has emerged, and how parallel architectures can be classified.

From 1986 to 2002 the performance of microprocessors increased, on average, 50% per year [7]. This impressive growth meant that software designers could often simply wait for the next generation of processor in order to obtain better performance from an application. After decades of rapid increase in CPU clock rates, however, they have now ceased to increase, due to what is known as the "three walls" [10]:

Power wall: Unacceptable growth in power dissipation requirements with clock rate.
Instruction-level parallelism wall: Limits due to low-level parallelism.
Memory wall: A growing discrepancy of processor speeds relative to memory speeds.

By 2005, most of the major manufacturers of microprocessors had decided that the road to rapidly increasing performance lay in the direction of parallelism. Rather than trying to continue to develop ever-faster monolithic processors, manufacturers started putting multiple complete processors on a single integrated circuit. [11]. Such processors are called multi-core processors. The term many-core processor is often used to refer to a multi-core processor with a high number of cores. To exploit the capabilities of these processors, parallel applications need to be developed.

There are two main classifications of parallel hardware, regarding the architecture and memory organization. Flynn's taxonomy [6], is often used in parallel computing to classify hardware architectures into four different categories. We are mainly concerned about MIMD, which stands for multiple instruction, multiple data stream. In this category, multiple autonomous processors simultaneously execute different instructions

on different data. They either exploit a single shared memory space or a distributed memory space. Other categories include SISD, SIMD and MISD.

Also, the memory of parallel architectures can be categorized into three major groups:

1. Shared memory: the memory is shared between processing elements in a shared address space.
2. Distributed memory: each processing node has its own local memory with local address space.
3. Distributed shared memory: a form of memory architecture where the physically separate memories can be addressed as one (logically shared) address space [7].

In parallel architectures, the memory wall is still imposing limitations, specifically to inter-processor communication, which can limit scalability. The main problems are: **latency** and **bandwidth**. Latency is the time between submission of a request until its satisfaction. Bandwidth is the overall output data-rate.

3.2. Adapteva's Epiphany architecture

Epiphany architecture defines a many-core, scalable, distributed-shared memory, MIMD parallel computing fabric. This section is a condensed overview of the architecture, based on Epiphany's architecture reference [3]. In this section some details of processing nodes, internal memory and inter-core communication will be discussed. A gnu based tool-chain that is used to develop applications for Epiphany. For more information on this architecture please refer to [3].

Each processing node is a super scalar floating-point RISC CPU that can execute two floating point operations and a 64-bit memory load operation on every clock cycle.

The Epiphany architecture uses a distributed shared memory model. Meaning that each core has a local memory, and has the ability to access the memory of all other cores using a global address space. The global address space consists of 32 bit addresses. Each mesh node has a globally addressable ID that allows communication with all other mesh nodes in the system. The higher 12 bits of the address space are used for the node's ID, and the lower 20 bits are used to address the local memory of each node. If the higher 12 bits are set to zero, the address will point to the local address space.

The local memory of each node is divided into 4 banks which can be accessed by 4 different masters simultaneously: **Instruction fetch**, **Load/Store**, **External agent** and **DMA** (Direct Memory Access, discussed below). This provides 32 bytes/cycle

bandwidth (64 bit/cycle per bank). To optimize for speed, the program should be designed such that multiple masters do not access the same bank simultaneously. This can be achieved for example by putting the code and input/output buffers of data in different banks. Manipulating the placement of program sections can be achieved by using section attributes (a feature supported by most modern linkers) which will be discussed in detail later.

The inter-core communication is enabled by the eMesh Network-On-Chip. It consists of 3 separate mesh structures, each serving different types of transaction traffic:

1. **cMesh:** used for write transactions destined for an on-chip mesh node. It has a maximum bidirectional throughput of 8 bytes/cycle.
2. **rMesh:** used for all read transactions. It has a maximum throughput of 1 read transaction every 8 clock cycles.
3. **xMesh:** used for transactions destined for off-chip resources or another chip in a multi-chip system.

As can be seen, inter-core writes are approximately 16X faster than reads. To lower the communication overheads, it is important that programs use the high write transaction bandwidth and minimize on-chip read transactions.

To offload the task of communication from the CPU, each Epiphany processor node contains a DMA (Direct Memory Access) engine. The DMA engine works at the same clock frequency as the CPU and can transfer one 64-bit double word per clock cycle. The DMA engine has two general-purpose channels, with separate configuration for source and destination. The DMA transactions will use the same eMesh infrastructure as the CPU.

3.3. Parallel programming paradigms

The main idea behind most parallel applications is splitting the work into smaller tasks which can be run concurrently on several processing units. Depending on the type of hardware, different approaches might be taken to achieve this [11]:

1. In **Concurrent programming**, a program is one in which multiple tasks can be in *progress* at any instant. (multithreaded systems are an example of this).
2. In **Parallel computing**, a program is one in which multiple tasks *cooperate closely* to solve a problem. (this is the focus of this thesis work)
3. In **Distributed computing**, a program may need to cooperate with other programs to solve a problem.

Among all of these, *Parallel programming* on a many-core MIMD hardware is the focus of this thesis. There are two main approaches for splitting the work in such applications: *Task parallelism*, and *data parallelism*. Data parallelism is useful when the size

of the data which needs to be processed is too large for the available memory, or when the processing is time consuming. The requirement to achieve data parallelism is that it must be possible to split the data into smaller independent parts. These parts can then be processed on different cores simultaneously to achieve speed-up. In data-parallelism, *the same computational work is performed on different parts of data concurrently*. Task parallelism on the other hand, focuses on distributing the computational work. If these computations work on data which do not depend on each other (at least at certain stages), the tasks can be executed in parallel. Otherwise, if the result of a work is necessary for the next one to start, the tasks has to be executed in sequence to form a pipeline-like parallelism. Both of these approaches can have benefits and result in speed-ups. A parallel application can take advantage of data parallelism, task parallelism or a combination of them. While designing a parallel application, we split the work into smaller units. We call each of these units a **task**. Tasks might be created to achieve data parallelism or task parallelism.

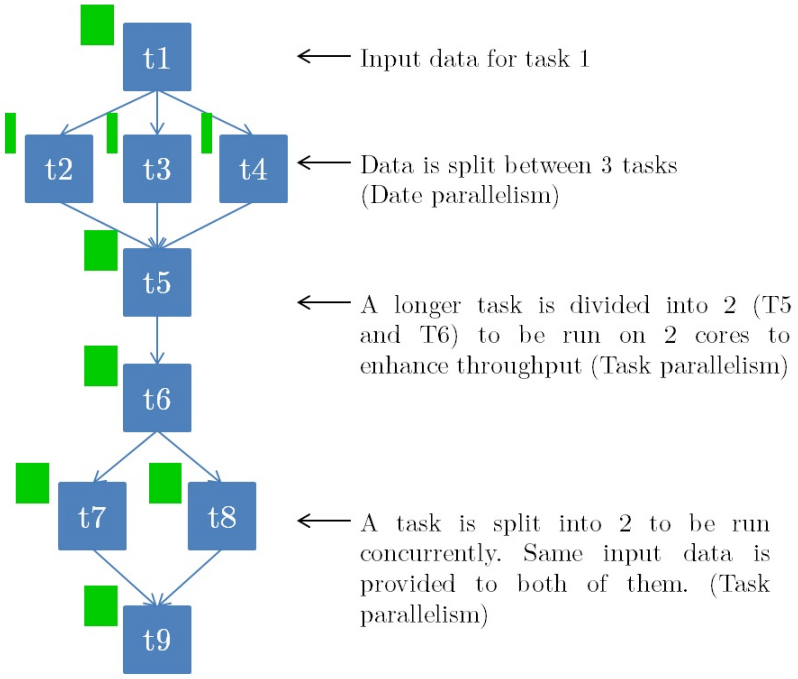


Figure 3.1.: Data flow in a parallel application exploiting both data parallelism and task parallelism. The color green represents data.

Parallel applications should be designed such that they have a good load balancing: All branches of every fork should have similar latency. In reality, it's almost impossible to achieve a perfect solution and there are always some differences in the latency of branches. Different latencies on different branches introduces a race hazard which needs to be dealt with: before the task that comes after a join can be initiated, it has to wait until the tasks on all branches are finished. One solution to this is using **barriers** (figure3.2). A barrier is a concept used for synchronization. Having a barrier after some tasks means that the subsequent tasks can not start until all previous ones (the ones coming before the barrier) have finished their execution and copied their data

to the subsequent tasks.

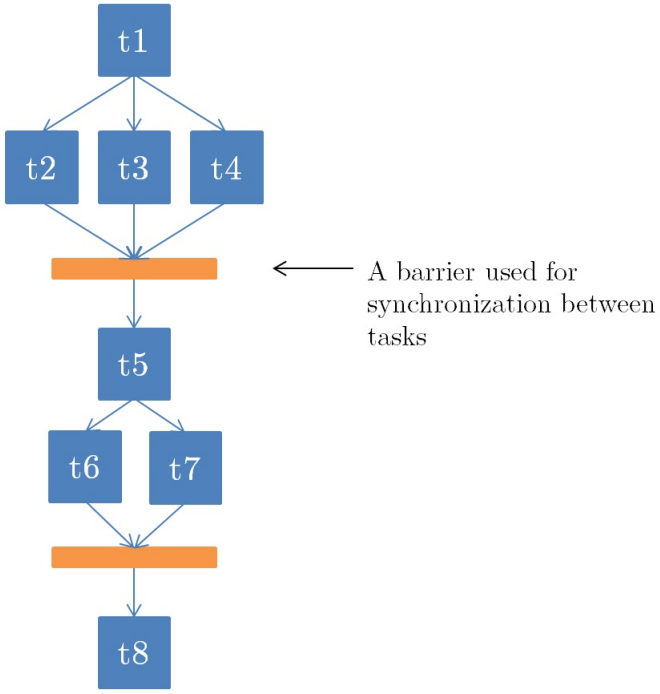


Figure 3.2.: Barriers

After splitting the work into smaller tasks, these tasks then need to be assigned to different cores of a many-core hardware. This process is called *mapping*. Each core of the hardware would then be responsible for the execution of one or more of the tasks. The tasks will communicate to each other and pass their processed data to the next task. If the next task is located on a different core, an inter-core communication of the data will happen.

The focus of this thesis work is replacing the communication overhead of data movement with a smaller (if possible) communication overhead of code movement at some parts of the graph.

3.4. BOS

The purpose of having an operating system is to manage available hardware resources. On a parallel hardware, the available resources among others include:

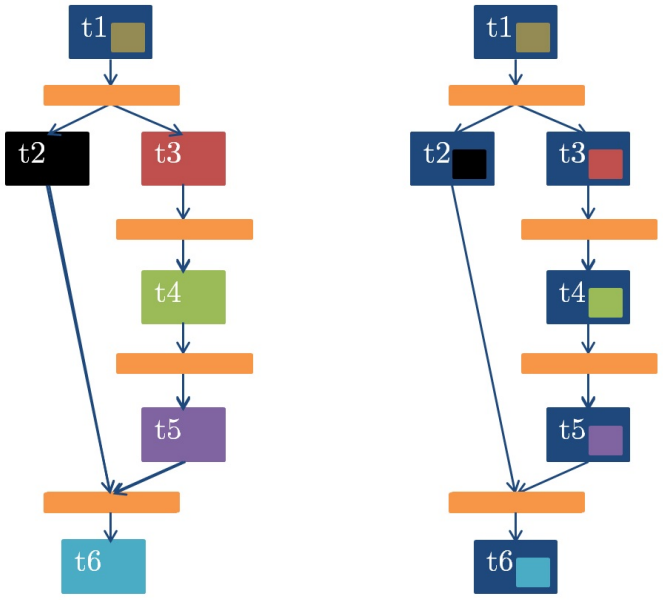
1. Computational power, which is distributed over multiple cores
2. The available memory

3. Inter-core communication channels

In embedded operating systems, due to limitations of memory size and computational power, usually the application and operating system are statically linked into a single executable image [4].

BOS (Basic Operating System) is an experimental operating system under development at Ericsson AB which is designed to operate in a very dynamic environment. It enhances mapping of computational support on a many-core processor and describing the flow of data between tasks in parallel applications. It supports *task parallelism* and *data parallelism* parallel application paradigms and provides accurate functionality for time measurement and benchmarking purposes.

In BOS applications are described using a graph, like the one in figure 3.4.a. In this graph, each node is a task (smallest work unit in parallel applications). As has been discussed, often in parallel applications there is a need for synchronization on different branches of the application. In BOS this synchronization is achieved using barriers. Because of some implementation details which we will not discuss, barriers in BOS need to be inserted after every task in the graph.



(a) (b)

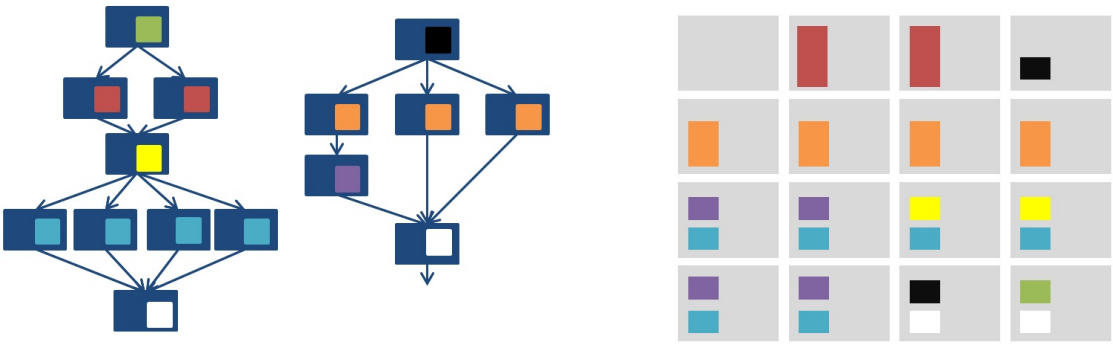
Figure 3.3.: (a) shows an example of a parallel application. Each box is a task (smallest work unit in parallel applications) and orange rectangles are barriers, inserted for synchronization. (b) is the same example with the difference that kernels (the functionality of tasks) are seperated from their respective tasks. In BOS, only kernels are mapped to hardware cores and the tasks are dynamically mapped to the cores which support their kernel during runtime.

The challenge that BOS is designed to overcome is to execute dynamic parallel applications, in which each batch of input data needs to be processed differently using a different application (figure 3.4.a). One way to approach this challenge is to load the whole application to the Epiphany chip for each batch of input data. This wouldn't be a very interesting solution since communication channel is quite slow and taking this approach will increase the latency of the application significantly.

In the domain that BOS is designed to operate in (LTE applications in radio base stations), the applications do differ; however, the tasks in these applications often share the same functionality (e.g. FFT, FIR, etc.). The only difference is the number of tasks and the order in which they appear. We can use this to our advantage by enabling the cores of the hardware to execute different kinds of tasks. In this way, the tasks of the application can be assigned to the cores during runtime and we won't need to reload the whole application every time. In order to do this, we need to differentiate between the functionality of tasks and the tasks themselves.

In BOS, the functionality of a task is called a kernel and it is differentiated from the task itself. The advantage of separating the kernels from the tasks is that we can statically map the kernels to the different cores of the hardware (figure 3.4.b), and let

the tasks be mapped to the cores dynamically on during run-time. This will eliminate the need for reloading the application to the hardware and reduces communication overhear.



(a) (b)

Figure 3.4.: (a) The parallel application's graph required to process each batch of data can be different. (b) Kernels are statically mapped to different cores of the Epiphany hardware. Each task will be assigned to a core which supports its kernel during runtime. The hardware will be able to execute many different graphs, as long as all required kernels are supported.

BOS is a distributed operating system and it exists on each core of the Epiphany chip. The mapped kernels are linked together with the operating system into a single image, and that image is loaded to different cores of Epiphany. As has been mentioned, Epiphany relies on an external host for loading the program and initiating it.

Once the program is loaded, the host will be able to send batches of input data to Epiphany in order for them to be processed. Each batch of data must be accompanied with a description of the parallel graph required to process the data. We simply call this description a graph. The graph can be different for every batch of data. The graphs designed by the application designer are described in XML format. The graphs are then sent to Epiphany, and are saved into the local memory of one of the cores. On the hardware, one core is responsible for receiving the graphs and initiating the application. This core is called graph receiver.

When the execution starts, each task will be mapped to a core that supports its kernel. This is a dynamic and non-deterministic operation which will happen during execution, and is carried out by BOS. After a task is executed on a core, BOS will be responsible to find a core to map the next task(s) to. The requirement for such a core is that it must support the kernel of the next task. After execution, the tasks are also responsible for copying their output data to the next core(s) (whose address(es) will be provided by BOS).

3.4.1. Graph description

In BOS, every parallel application is described using a directed graph, like the one found in figure 3.5, in which nodes represent either a tasks or a barriers, and the directed edges will represent the direction and path of the data-flow. Graphs are described in XML format by the application designer. On Epiphany they will be stored as a linked data structure containing the information of tasks and barriers.

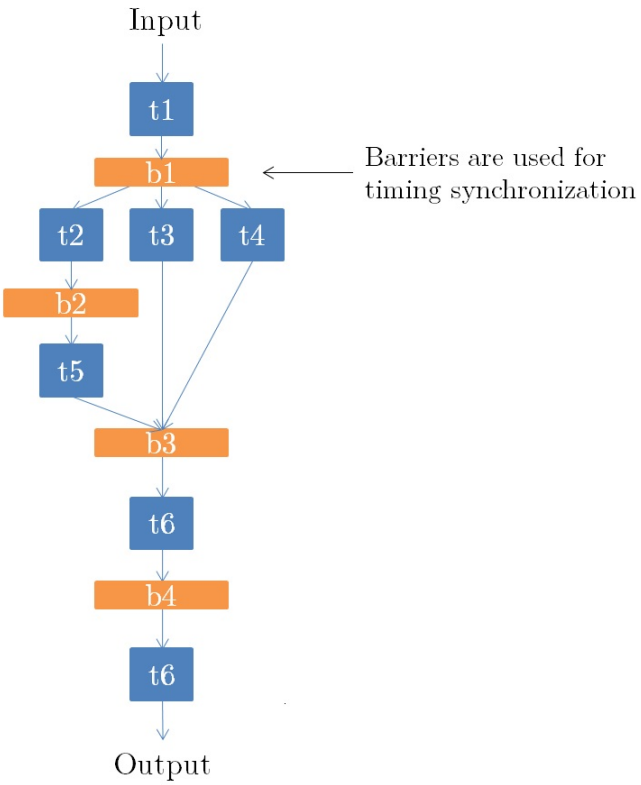


Figure 3.5.: High level graph representation

Tasks:

As was discussed before, in parallel applications the work is split into several work units. In BOS, these work units are called tasks. Each task is partly responsible for computing the final result. The task type describes which kernel it must execute and which barrier it points to. It also contains a unique name for this task. The tasks are described in XML format by the application designer. A sample task element in XML is shown in figure 3.6. The attributes that define a task are:

1. **Name**, which is used to differentiate between the tasks. Every task must have a unique name.
2. **Kernel**, which signifies the kernel of the task. Several tasks can run the same kernel, but not the other way around.

3. **Barrier**, which is the name of the barrier coming after the task. Several tasks might flow into the same barrier. There must be a barrier after every task (unless it is the final task of the graph) and tasks can not directly point to other tasks.

```
<task>
  <name>t1</name>
  <kernel>k1</kernel>
  <barrier>b1</barrier>
</task>
```

Figure 3.6.: The XML representation of a task

A task can be mapped to any core which supports it's kernel. This mapping is dynamic and is done during run-time. It is the responsibility of the BOS to find a core which supports the kernel of each task, and it is decided during run time in a non-deterministic manner. Once a task is mapped to a core, its input data is copied to that core and its kernel (whose code already exists on the core) will be initiated.

Several tasks can come between two barriers, and they might require the same kernel. We call these simultaneous kernels. During application design, it is vital to ensure that the cores will have support for enough number of simultaneous kernels.

Barriers:

As was described before, barriers are synchronization points. The task coming after a barrier will not start until all tasks before the barrier have finished execution. In BOS, every task will point to a barrier. Several tasks can point to one barrier, and this is what makes it possible to join multiple branches of the graph without any synchronization problems. Similarly, several tasks can come after a barrier, and this enables application graphs to have a fork. The attributes used in the XML file to describe the graph are:

1. **Name:** Similar to tasks, each barrier must have a unique name.
2. **Task:** The task(s) which come after the barrier. Several tasks can come after the barrier. For each one of them, a new $< task >$ attribute is added, with the task's name as its value.

```
<barrier>
  <name>b1</name>
  <task>t2</task>
  <task>t3</task>
</barrier>
```

Figure 3.7.: The XML representation of a barrier

Kernels:

In BOS, a kernel is an encapsulation of the operation the task has to perform (e.g. FFT, Filtering, etc.). Every task can have only one kernel, and multiple tasks can use the same kernel. Kernels are the only part of the application which are statically mapped to the Epiphany cores. Tasks are mapped dynamically to the cores during run-time, and they can be mapped to any core that supports their kernel. This is what enables BOS to run different shapes of graphs, without reloading the program on the cores. BOS code and kernels are statically linked together and mapped to the hardware cores. Several cores of the hardware can support the same kernel, and, as long as memory limitation allows it, several kernels can be supported by a single core (figure 3.4). Kernels provided to BOS, must be written in the C programming language and provide 2 interface functions:

1. Kernel run
2. Move data

The kernel run function (figure 3.8) is called when the task is initiated. All tasks expect the input data to be in place before they are initiated. A pointer to the I/O area is provided to the kernel run function by BOS.

```
void kernel01_run(void* io_p);
```

Figure 3.8.: Kernel run function interface

The move function, has the responsibility of copying the output data of a kernel to the destination core(s). In some cases, several tasks come after a barrier and the data has to be split in certain way. An array of pointers is provided to the move data function, which includes the I/O area of the cores supporting consecutive tasks. The number of these cores is also passed as an argument. The interface of a sample move function is shown in figure 3.9. In case the data needs to be split, it is the responsibility of the application designer to design the move function in a way that does so.

```
void kernel01_move_data(void** io_pp, int nmb);
```

Figure 3.9.: Move data function interface

As was mentioned, each core can support one or more kernels. Kernels are also associated with a unique number, that is called kernel id. The ids of the kernels that are supported by each core, are put in a variable in the local memory of the core, which is exposed to all other cores. This way, every core can check which kernels are supported by the other cores.

3.4.2. BOS Operation

When a program is loaded to the hardware and the cores are initiated, each core will go through an initialization stage. After that, Epiphany is ready to accept input data together with the graph information describing the interconnect of tasks and barriers. One core on the Epiphany chip is responsible for receiving graphs which is called the graph receiver. This core will receive and maintain the graph data.

Every core of the hardware has a few states which indicate whether the core is busy or ready to accept new tasks. In addition, every core supports one or more kernels. The state of the core and the kernels that it supports are accessible by other cores. Every core of Epiphany can look up in the memory of other cores to find this information.

After the graph is received, the receiver core will first look into the graph information of the first task to check which kernel it requires. It will then start polling every core of the hardware to find one which supports that kernel. After finding one, it will change the state of that core and reserve it. Then, it will copy the input data into that core's I/O area. Finally, the core's task pointer is changed to point to the first task (task is assigned) and it is initiated.

All cores at their idle state, are polling the state variable which can be changed by other cores (this is done when they are assigned a task). When the state is changed and the core is assigned a task, it will look up the task information in the graph receiver core's memory. There, it can find the kernel id of the task. All cores expect the input data to be in place when they are assigned a task, since it is the responsibility of every core to copy its output data to the consecutive task's core. Given that the input data is in place, and the kernel id is known, BOS can now call the respective run kernel function. It should be emphasized again that a core might support more than one kernel.

After the run kernel function executes, control is transferred back to BOS at the local core. At this point, BOS will check the barrier pointer of the task (section 3.4.3). If the pointer has a *NULL* value it means that the end of graph has reached, since every task is required to have a barrier after them, unless it is the terminating task (end of graph). In case the there is a value in this pointer, BOS will look into that address to find the barrier information. At this point a barrier is reached and the following need to happen:

1. For every task coming after the barrier, a core must be found which supports its kernel, and they must be reserved.
2. Output data of all tasks coming before the barrier must be copied to those cores.
3. Reserved cores must be initiated.

At this stage if more than one task comes before a barrier, multiple cores can end-up searching for a core that supports the next task's kernel. This can cause some problems: it will put more communication traffic on the network which can result in extra latency

for other cores. Furthermore, it is difficult to keep track of which kernel supports are already found when having several cores searching for them. Therefore, only one of the cores must carry out this search and assign new tasks to other cores. The first core that reaches a barrier is designated for this task. For the cores to find out if they are the first to reach a barrier, there is a flag in the barrier type. The first task (the core executing it) which reaches the barrier will find this flag to be zero and it will raise this flag. This flag is mutex protected.

The steps that will be taken after a barrier is reached in more detail are:

1. If the core is the first one to reach the barrier, it has the responsibility of looking for support for the kernels of the succeeding tasks. To do this, it will discover all tasks coming after the barrier and check their kernel number. It should be emphasized that this information exists in the linked data structure of graph, residing on the graph receiver core. The core will then start looking for kernel support on cores which are not busy. When found, it will reserve these cores by changing their state and it will *write the coordinates of these cores in the task members of the linked data structure*. This is so that the other cores running other tasks know where to copy their data to.
2. Regardless of the order in which the cores reach the barrier, every core must copy its output data to the coming tasks. It will try to do so by looking at the core_id of the task. If filled, it means that the core who had the responsibility of finding support, has found the cores and has written their address in this field. If this field is empty, it will wait for it to be filled. After fetching the coordinates from this field, it will call the *move data* function of the kernel. When calling this function, BOS will provide pointers to the I/O area of these cores. Once again, because of characteristics of Epiphany architecture, it is much more efficient if the data which is being passed it "written" by the previous core, instead of having the next core reading it.
3. After copying the data, the core will check if it is the last one to finish. If this is the case, it will initiate all reserved cores by changing their state.
4. Finally, the core must change its own state and go idle to be ready for future tasks.

All cores are polling their state variable while they are idle. When a task is assigned to them, same steps will happen: run the kernel, reach barrier, find support, copy output data and go idle again.

3.4.3. Internal representation of graphs

On Epiphany, currently one core is responsible for receiving graphs. Once the graph received by this core, it will be converted into a linked data structure containing the information about the tasks and barriers.

One difference between the graph representation at a high level and the internal representation of the graphs is that at a high level, each task points to a barrier and barriers point to the task(s) which come after them. Such approach is not memory efficient on the Epiphany, since the number of tasks which come after a barrier is not fixed, and it's not a good idea to reserve space for the maximum number of tasks which "might" come after a barrier in the data structure. A more efficient approach is to let the barrier point to only the first task which comes after it, and let the tasks point to their neighbor tasks which also come after the same barrier. We call the tasks that come after the same barrier *parallel tasks*. This way, no matter how many tasks come after a barrier, they can point to each other until the last task, and the last one can have null pointer as the parallel task pointer.

Furthermore, a task type contains an id, the type of the kernel it needs to run to process its input data and the core_id. The core_id is not known when the graph starts execution; After each task is run, the core running it will look at the next coming task's kernel_id and then it will try to find a core which supports this kernel. If found, it will reserve that core, and put that core's id in the core_id data member. This will used later for moving data to that core.

Each task essentially contains a pointer to the barriers it runs into. It may also contain a pointer to a parallel task. If two or more tasks come between two barriers they are called parallel tasks.

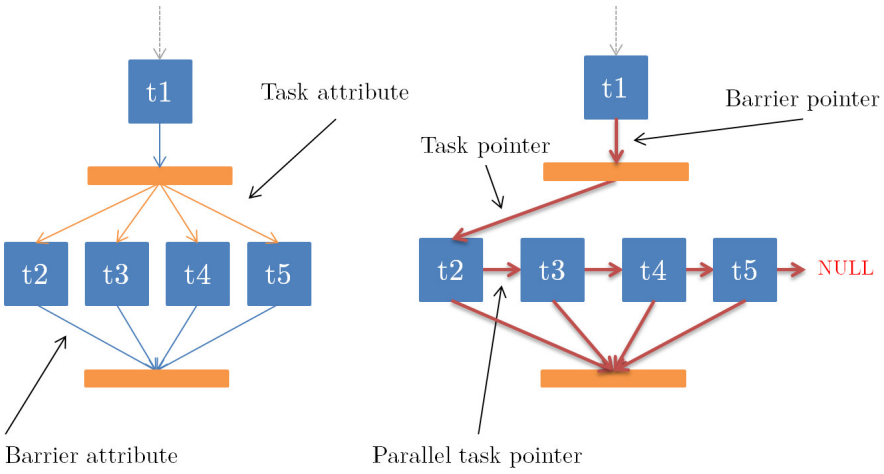


Figure 3.10.: Internal representation of graph versus the XML representation. In xml representaion, each barrier points to all subsequent tasks. In the internal representation however, a barrier will point to only one of the subsequent tasks and each task will contain a pointer to the neighbouring parallel task.

The task type:

```
typedef struct task {
  unsigned short id;
  unsigned short kernel_id;
  unsigned short core_id;
  struct task *parallel_task_p;
  struct barrier *barrier_p;
} task_t;
```

Figure 3.11.: The data-structure representing a task

1. **id:** Which is used to differentiate between the tasks and is useful for pointing to tasks in barriers. It will be ignored later when the linked list is created.
2. **kernel id:** This is the kernel type which will need to process the data of this task. Please note that there is no explicit mapping constraints given here.
3. **core id:** This will represent the id of the core to which the task is assigned.
4. **barrier pointer:** This is the barrier that comes after the task.
5. **parallel task pointer:** Points to parallel tasks after the barrier.

The bearer type:

```
typedef struct barrier {
  unsigned short id;
  unsigned short semaphore;
  unsigned short reached;
  unsigned short dummy;
  struct task *task_p;// pointer to 'first' task to follow
  e_mutex_t lock;
} barrier_t;
```

Figure 3.12.: The data-structure representing a barrier

1. **id:** Just like task id, this is used in the XML file and after the linked list is constructed it is ignored.
2. **semaphore:** This is initialized to the number of tasks which flow into the barrier. Tasks use this to get to know if all parallel tasks are finished.
3. **reached:** This is set to 1 once the first task reaches the barrier. This way, other tasks before the barrier will know if they are the first task to reach the barrier or not.
4. **task pointer:** This points to the next task which comes after the barrier. In case there's more than one task, the rest will be pointed to using the parallel task pointer in the task data-structure.

On the Epiphany at least one core must support graph reception. The host will provide the interface graph to this core. During the execution of the program, when a barrier is run into by a core, the core will need to access the information regarding the tasks which come after the barrier. It is important that the data structure is implemented in an efficient way so that accessing these members doesn't introduce a big overhead.

For these reasons, the interface graph is translated into a linked data structure which holds the information about the tasks, barriers and the interconnect of them. To have memory efficiency and keeping the size of these data structures constant, there's a slight difference between representing a graph at a high level and the representation which exists on the hardware: instead of having the barrier pointing to all of the tasks which are coming after it, the barrier only points to the first task. This task will then contain a pointer to the first (if any) of the tasks which are coming after the same barrier. This makes it possible to have constant size task and barrier types and save memory space. This is illustrated in figure 3.10.

The graph, which only describes the interconnection between the tasks and barriers and contains no input data, will go through 2 translation stages in the host before it is sent to the Epiphany chip. On Epiphany, one core must have the ability to receive graphs and is called the graph receiver. This core will take the graph and do another translation stage on it until it finally turns into a linked data structure representing tasks and barriers. Each task will point to the subsequent barrier and each barrier will point to subsequent tasks. This information (the data structure) will be held on the graph receiver core until the end of the execution.

Other cores of the hardware, can access the graph info to find out about the next task's kernel number. As was mentioned before, there is no static mapping for the graphs and there are no constraint on where the task has to be executed. What happens after hitting a barrier is that the core which finished execution of its task (the first one to finish) will look for support of the kernel of the next tasks which come after the barrier. As long as there is support for their kernels, the consecutive tasks can be executed on any core.

3.4.4. Core states

There are four possible stated for a worker core: init, idle, reserved and busy. Every core will enter the init state at start up. At this stage, some initialization functions will be run once and then the core will switch state to idle. The initialization functions include:

1. Initialize timers (for trace functionality)
2. Allocate space for trace_buffer and core_info, and and put a pointer pointing to them in the corresponding positions in BOS pointer area

3. zero out the pointers related to core to core interactions in BOS pointer area
4. on core zero, allocate space for graph_area and interface graph_area and put a pointer to them in BOS pointer area.

After entering idle state, the core is ready to to accept tasks. Each core, holds a constant which indicate its supported kernels. This constant is accessible by other cores. When other cores are finished with their own task and are searching for cores which support the consecutive task of the graph, they will check every core, and they start by checking this variable to see if this core supports the needed task. Then, they will also check the state variable to if the core is idle. If it is, they will reserve the core by changing its state to reserved. After the data is moved to this core, the task will be assigned to it by pointing it and it's state will be changed to busy. The core state variable is mutex protected.

3.4.5. Trace functionality

One of the features of BOS is the trace functionality it provides. An important factor, which determines usability and accuracy of any trace functionality, is the amount of overhead it introduces in the total execution time: putting any probe in the code will result in execution of some extra cycles.

To keep this overhead low, it was decided to keep the trace buffers inside the local memory, as any access to external memory would cost a lot of cycles due to the slow connection and is not deterministic.

The Timers are initiated during core start up (bos init) and they will keep counting down. To trace any type of event, the inserted probe in the code would just read the timer value at that specific moment and dump the timer value together with event type into the trace buffer. This task is carried out by a trace function, which takes the event type as its argument.

Events are written to the trace buffer one after each other as they occur. These events might not have the same length. For every event, a "trace_header" is inserted into the trace buffer. The data structure for the "trace_header" is shown in 3.15. The data that it contains are: the event type and the time it has occurred. Depending on the type of the event, it might need to carry some extra information as well, this information are written to the trace buffer right after the trace_header. The length of this information will be saved in the length field in the "trace_header" data structure. This will be useful when data is being extracted from the trace buffer.

There are two types of events: Events which do not have a duration, and events which do have a duration (start time and finish time are important). There is a code corresponding to every event that is saved in the "type" field and later will be interpreted by the host.

```
// Host buffer, hosting trace data from one core
typedef struct trace_buffer {
        uint16_t write_offset;
        uint16_t read_offset;
        uint8_t jam;
        uint8_t buffer[TRACE_BUFFER_SIZE];
} trace_buffer_t;
```

Figure 3.13.: Trace buffer header

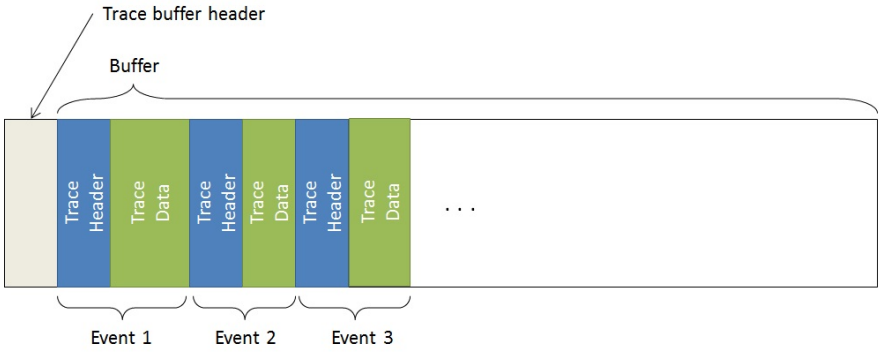


Figure 3.14.: The trace buffer
Each entry has a trace header, and a varying size of data.

```
        // Formatted trace header
        typedef struct trace_header {
                uint8_t length;
                uint8_t type;
                uint16_t time_stamp_lo;
                uint16_t time_stamp_hi;
        } trace_header_t;
```

Figure 3.15.: Trace header type. Each entery of the trace info has this data structure.

3.5. Compilers

In this section, we will review some basics of compiling, linking and object file representation. Some important principles regarding executable files and linker techniques associated with them which were useful towards achieving a solution for this thesis work are also discussed.

Compiling is the process of transforming a computer program written in a usually high level language (in our case C) into an executable file. Compiling stages, as shown in figure 3.16 start with transforming every C source file into assembly code. The assembler will then transform assembly code into object files. These object files are then linked together using the linker to form a single file.

Each object file consists of object code plus some extra information which will be used in the process of linking. The file format currently used for object files is ELF, which is also used for executables, shared libraries and memory dumps.

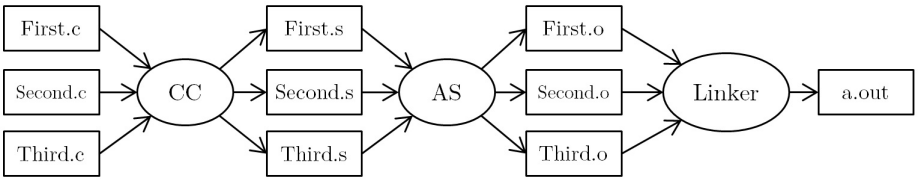


Figure 3.16.: Compile stages

When an executable file is created, different parts of the code and data are layed out in the address space in an organized way. Usually there are 3 abstract elements in a computer program:

1. .text
2. .data
3. .bss

.text is the program code. The instructions which will be executed. **.data** is the initialized data of the program. In C, that would be initialized static variables and initialized global variables. **.bss** which stands for (Block Started by Symbol), in the uninitialized data, i.e., uninitialized static variables and uninitialized globals.

Every object file usually has these 3 segments, however, bss does not actually take any space in the output file, although space is allocated for it. This is to save memory.

A computer program, in addition to these, has two additional dynamic segments: **stack** and **heap**. The typical layout of a computer program is shown in figure 3.17.

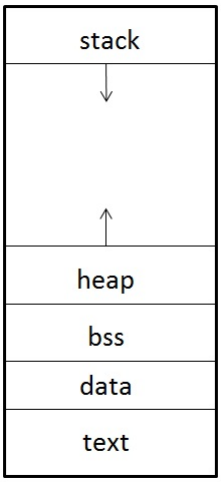


Figure 3.17.: Typical layout of program segments. The heap starts after data segments and grows towards higher addresses. The stack is allocated at the end of the address space and grows downwards during program execution.

3.6. The linker

With today's programming tools, programmers are allowed to use abstract symbols for different parts of the program (function names, variable names, etc). An object file, may contain references to external symbols (other object files). It may also contain unresolved references to internal symbols: when object files are created, the assembler will assume that the address space starts at 0 for each object file, but this won't be the case when these files are combined together to form an executable since each part of the program might end up in a different address of the memory. The job of the linker, simply put, is to combine all object files and replace all symbols with concrete addresses in the memory.

After the second stage of compiling, several object files are produced. It is then the job of the linker to take all these object files and produce one single executable file out of them.

To do this, along with the object files, the linker also needs some extra information which include: memory layout of the output and information regarding mapping of different program sections to the output file. This information is provided to the linker using a *linker script*. Linker scripts for the gnu linker are written in the *command language*, details of which can be found in [13]. For all targets, there exists at least one default linker script, but the programmer can provide a customized linker script to gain more control. If provided, the new linker script will override the default one.

The process of linking involves several steps. The linker will first scan all input object

files and identify all of their symbols and program sections. Then it will read the linker script to map input sections to the output sections by name. After that, it will assign addresses and finally it will copy input sections to output sections and apply relocation.

At a very high level, the GNU linker follows these steps [12]:

1. Walk over all the input files, identifying the symbols and the input sections.
2. Walk over the linker script, assigning each input section to an output section based on its name.
3. Walk over the linker script again, assigning addresses to the output sections.
4. Walk over the input files again, copying input sections to the output file and applying relocation.

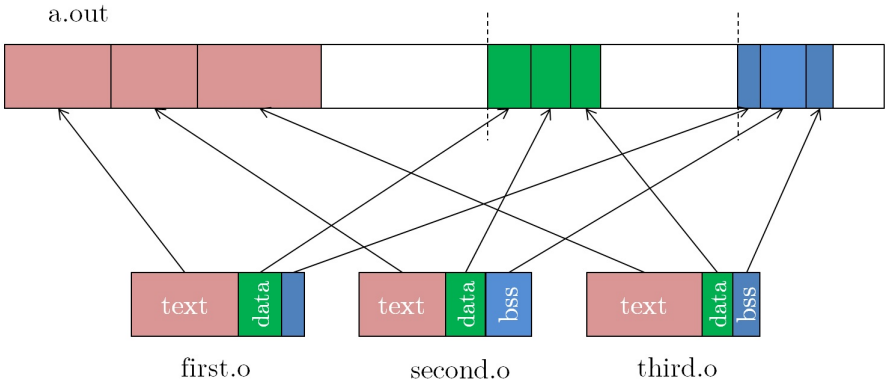


Figure 3.18.: Linking different object files

It is important to mention that when the linker combines different object files, it will combine different segments of them, that is, by default the final executable will only have one .text segment, one .data segment and one .bss segment. This is illustrated in figure 3.18. As can be seen, similar segments from different files are combined together. It must also be pointed out that the linker will not shuffle sections around to fit them into the available memory regions, and the sections will appear in the object file in the same order that they appear in the C file [13].

Now we will review some of the concepts assotiated with the linking process:

Loading:

The task of a loader is to copy the program from a secondary storage into the processors main memory. In case of Epiphany, this is performed by an external host by writing to each core's local memory and then starting them. On some more complicated systems, loading might involve handling virtual memory, storage allocation or

several other tasks.

Position independent code:

Position independent code is a type of code which will not change regardless of the address at which it is loaded [8]. This means it can be places anywhere in the memory and execute properly regardless of its abdolute address. This is in contrast with relocatable code where the code needs some modification after being moved in and before it can be executed.

Relocation:

The object files that are generated by assemblers, usually start at address zero which also contain lots of symbols which are referenced inside the object file or are referenced by other object files. Relocation, is the process of assigning non-overlapping load addresses to each part of the program, copying the code and data to those addresses and adjusting the code and data to reflect the assigned address.

Symbol resolution:

When multiple object files form a program, the references from one program to another are made using symbols. After relocation and allocating space to every module, the linker should also resolve the symbols meaning that it will replace all of the symbols with actual addresses in the memory in which the data or the routine represented by that symbol resides.

3.7. Dynamic linking

In dynamic linking, much of the linking process is deferred until run-time. While it has several advantages regarding maintenance of shared libraries, it introduces many disadvantages which in case of this thesis work outweigh the advantages significantly:

1. It introduces significant run-time cost since most of the linking process has to be done after a body of code has been copied in. This would introduce a new overhead which might counterbalance or even exceed the speed-up potential of moving smaller sized code instead of data.
2. Dynamically linked libraries are larger in size, since they include an additional symbol table. The code which performs the linking process also takes memory space. On a hardware like Epiphany with current limitations of memory, these facts make dynamic linking much less attractive as a solution.

4

Evaluating code movement vs data movement

The research question in this thesis work is to find out whether or not it can be beneficial to move the code instead of data. As was mentioned in chapter 1, we will focus on dynamically delivering the code to cores where the tasks come in a sequential order after each other and not in the fork-join constructs.

To answer the research question, one possibility is to construct tasks sequentially, map all of them to one core, and deliver the kernels to that core as the tasks run one by one. We can describe sequential tasks using the current graph description of BOS; however, if tasks are to be executed sequentially on the same core, BOS expects the kernels of those tasks to be initially mapped to the core, and there is currently no way to instruct BOS to first deliver a kernel to the core and then execute it.

In order to overcome this problem, we can introduce a new form of interconnection between the tasks which we call "sequential task" (figure 4.1). Tasks can now point to either a barrier (like the current flow of control in BOS) or, as the new interconnection form suggests, they can directly point to another task (sequential task construct) without any barriers in between. Furthermore, the sequential chain of tasks can be mapped to one core, and having this construct would signal the BOS that the kernel for each consecutive task needs to be delivered to the core before it is initiated.

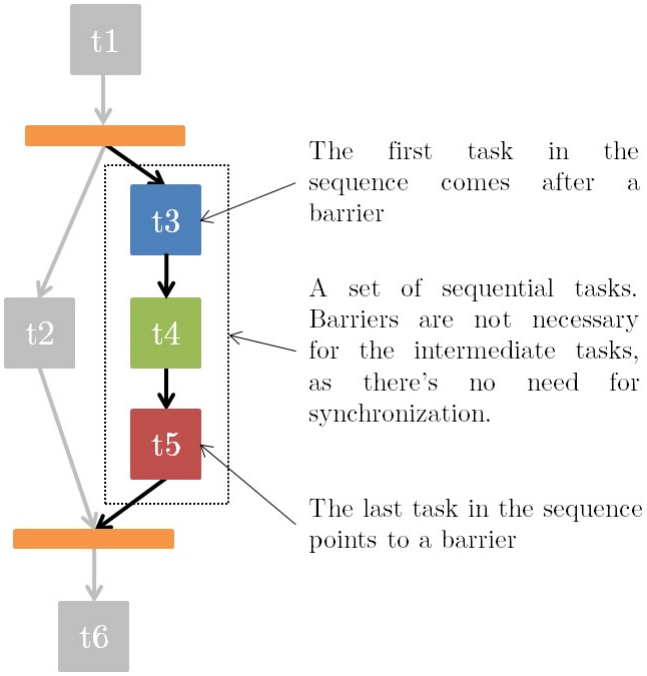


Figure 4.1.: The new type of tasks

The main purpose a barrier serves, is to act as a synchronization point where multiple tasks run into one task; however, if we construct the tasks sequentially, synchronization would no longer be necessary between the tasks, as there are no multiple branches joining together. This can also have performance benefits, since the platform overhead of reaching barriers is eliminated (details of BOS operation when a core reaches a barrier is described in section 3.4).

To enable running sequentially constructed tasks on a core, we need to store the kernels ready for execution outside the core at a place from which they can be delivered to the core with the smallest communication overhead possible. The possibilities for a storage place for the kernels are discussed in the next section.

Later in this chapter, we will also discuss how the new solution is integrated into BOS, how a piece of code can be made relocatable and how the memory layout is designed to support moving and delivering of kernels. To evaluate the solution, a benchmark scenario was constructed which is presented later in this chapter, along with the results and their analysis.

4.1. Storage cores, why are they needed?

In order to deliver kernels to the cores which run sequentially constructed tasks, the kernels need to be stored at some place and get delivered to the core(s) which are

executing sequential tasks. The options for a storage place on the Parallela board are the external shared memory and the local memory of neighboring cores. Among these alternatives it was decided to store the kernels on the local memory of a neighboring core. From now on, we refer to these cores as "storage core" (figure 4.2). It was also chosen to put the responsibility of delivering the kernels on the storage cores. These decisions were made for the following reasons:

1. On the Parallela board, the connection to the shared memory is extremely slow and this can increase the communication overhead significantly. Instead, storing the code on a storage core, lets us take advantage of the faster inter-core communication.
2. As was mentioned in section 3.2, on the Epiphany architecture, writing to non-local memory is much faster than reading from it. Having a storage core responsible for delivering the kernels, will allow us take advantage of this fact and have the kernels delivered to the requester core while being written, instead of having the requester perform memory reads to bring in the new kernels.

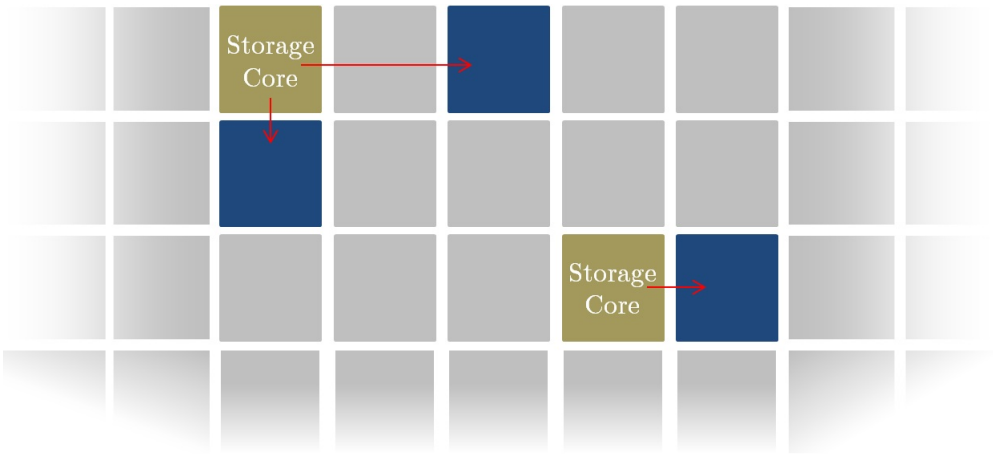


Figure 4.2.: Storage cores can store kernels and provide them to the cores which execute tasks sequentially. Spacial locality can lower the load on the communication network.

At their idle state, storage cores will be waiting for request messages from other cores. Once a request is received, the storage core will initiate a memcopy operation and supply the requester core with the kernel.

The number of storage cores, the kernels stored on them, and their position would be up to the application designer to decide on. Obviously it would be of great advantage if the storage core(s) are put very close to the core(s) which are executing sequential tasks, to reduce the latency and the traffic on the network. Furthermore, it should be pointed out that it is possible that a core has graph reception and kernel storage functionalities at the same time, as long as the memory limitations allow that. With the current implementation, storage cores will use CPU writes to deliver kernels to

other cores. A possible future optimization is to use the DMA channels of Epiphany, to offload the work from the CPU.

4.2. Changes to graph representation

In order to integrate the new feature of sequential task execution into BOS, the first step is to represent this concept in the graph representation. The first addition will be in the high level representation of the graph (which is the XML representation). A new attribute (seq_task) is added to the task element (figure 4.3) to let the application designer indicate tasks which will come in sequence:

```
<task>
  <name>t1</name>
  <kernel>k1</kernel>
  <seq_task>t2</seq_task>
</task>
```

Figure 4.3.: seq_task attribute will now indicate the successor task which comes in sequence.

The *seq_task* attribute will indicate that the tasks should come in series. As was mentioned in the beginning of this chapter, with the new type of connection barriers won't be necessary anymore. A task can now have either a barrier attribute or a sequential task attribute to describe which task it precedes, or none of them in case it is the ending point of the graph. A graph with sequential tasks is illustrated in figure 4.1.

A similar change can also happen in the internal representation of the graph: a new field can be added to the task type to point to other tasks. This new field is called *seq_task_p* in the task data structure (figure 4.4. When the graph is received and translated in the graph receiver core, the seq_task_p can point to the task which comes in sequence. This will enable BOS to recognize sequential inter-connects.

```
typedef struct task {
  unsigned short id;
  unsigned short kernel_id;
  unsigned short core_id;
  struct task *parallel_task_p;
  struct task *seq_task_p;
  struct barrier *barrier_p;
} task_t;
```

Figure 4.4.: seq_task_p field is added to the task data structue to point to sequential tasks

4.3. Changes to BOS operation

As was discussed before, in this thesis work we have focused on moving code for that tasks which come in a sequential order. The sequential tasks can be mapped to one core, and the kernels of the tasks can be delivered to it one after another. Since the data will reside on the same core and it is not moved, we will exchange communication overhead of data movement for code movement.

The core to which the sequential tasks should be mapped are chosen by the application designer. The sequential tasks will be represented in the XML format and will be further translated to internal representation of the graph. This information will reside on the graph receiver core. Also, the kernels of the tasks in the sequentially constructed chain will be available in the local memory of a storage core (figure 4.5).

The cores to which sequential tasks are mapped, will initially support the kernel of the first task of the sequential chain, and they will expose this information. From other cores' point of view, they have no difference: when BOS reaches a barrier that is situated before a sequential chain, it will start to search for support for the first task's kernel, and it can easily find the intended core since it already supports the first task's kernel. When this core is found, the input data will be moved to it and it will be initiated, just like normal BOS operation which has been described before.

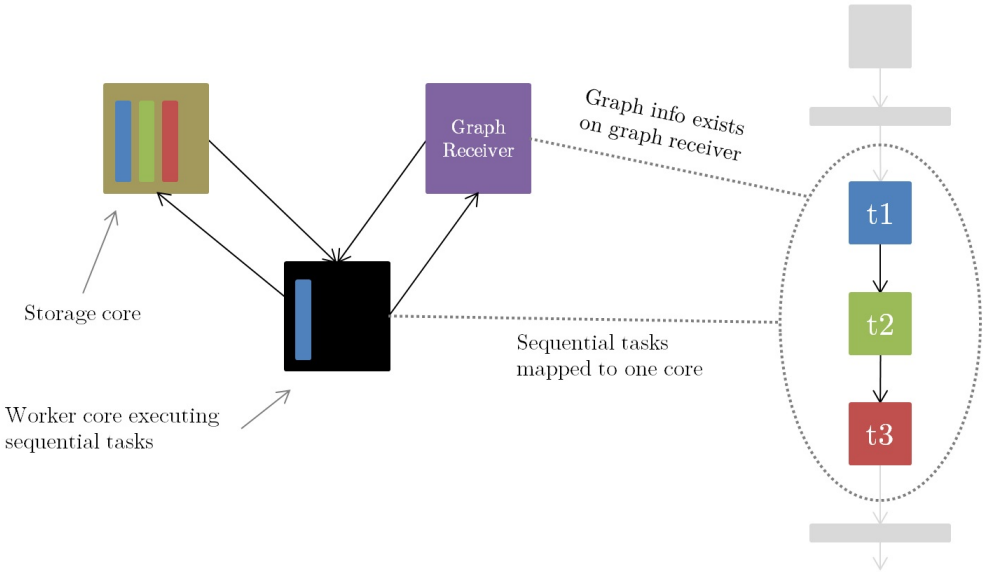


Figure 4.5.: Three different types of cores: Storage core, graph receiver core, and the worker core executing sequential tasks. The latter, will initially support the first task of sequence, and this information will be exposed to other cores; this way, after the barrier before the first task is reached, this core can be searched for and be found. After assignment of the task, this core will check the graph information on the graph receiver core after every task, and as long as a sequential task exists, it will fetch the task's kernel id and request it from the storage core. The storage core would then write the kernel to this core, and the core will continue the execution. Once a barrier is reached, the core will continue normal BOS operation (move data, initiate next core). Before going to idle state, the first kernel (in this example the blue one) will be post-fetched from the storage core so that the core can start execution for the next set of data without any delay.

After the execution of the first task, BOS will check the graph information which resides in the graph receiver core. A small addition to operation of these special cores is that they check for a sequential task pointer before they check for a barrier (other cores which won't execute sequential tasks will not perform this check, to avoid extra unnecessary traffic on the network). If a sequential task pointer is present, it's data will be fetched and it will be set as the current task of the core.

The worker core then needs to receive the kernel of the new task. To make this happen, it will send a message to the storage core which has the code for the new kernel. The message consists of the number of the requested kernel, and the address of the core requesting it. The storage core will then initiate a memcopy operation, to copy the kernel code to the core which has requested it. After the memcopy is finished, it will notify the requester core.

Normally, before a new task is started BOS calls the move data function to deliver

the data to the core which will execute that task. But in this case, the data doesn't have to be moved since the next task will be executed on the same core. To avoid any communication overhead for internal data movement, multiple buffering can be used. Before the next task is initiated, pointers to the input and output data can be swapped and be provided to the new task.

After the delivery of the new kernel and swapping pointers, BOS will call the kernel run function through the wrapper, passing through the pointers to the current input and output areas. After the kernel is executed, BOS will check the task information again for a new sequential task. If there is a pointer to a sequential task, this procedure will repeat until the last task in the chain is executed.

After the execution of the last task, BOS will not find a pointer to a sequential task anymore. Instead, if the task is not the last task of the graph, it will find a pointer to a barrier. At this point, ordinary BOS procedure will execute (the ones described in section 3.4) to find support for the next task, copy the data and initiate the next core.

When support for the next task is found, the move function will be called using the wrapper, with a pointer to the current output area and a pointer to the input area of the next core. The wrapper will then call the move data function and pass these arguments to it. The data is then moved to the next core and that core is triggered.

After all the tasks in the chain are executed, the core will have support for the kernel of the last task in the chain. This can cause a delay when the next set of data is assigned to the core, because then the kernel for the first task of chain should be requested to be brought in. To avoid this delay, after the core has finished its job and initiated the next core, and before going to idle state, it can post-fetch the first task's kernel by sending a request to the storage core. Obviously the storage core must store the kernels for all of the tasks in the chain, including the first one.

4.4. Manipulating the memory layout

The aim of this section is to address some of the technical issues which need to be dealt with before achieving a solution. If a body of code is to be moved and executed, there will be three major considerations:

1. The moved code has to be relocatable and self-contained
2. Space for the code needs to be pre-allocated
3. We need a way of calling the code

Relocation and being self-contained:

Having the code relocatable is not difficult. The generated code for Epiphany is position independent meaning it can be executed from any address in the memory, as long as the 4 Byte alignment restriction is considered. But having the kernel code to be self contained can be an issue: as was explained in section 3.6, while the linker is generating a final executable file, it will combine all similar segments together (figure 3.18). This means that different segments of a kernel (i.e. the code and the data associated with it) will be placed in three different segments: the code will be places in the .text segment, the initialized data will be placed in the .data segment and the uninitialized data will be allocated in the .bss segment. If we try to move the kernel, we would have to copy the code and also the data which is situated in a different place. Otherwise the data will be lost (figure 4.6.a). Also, the code contains several load/store operations to work on the data which are assumed to be at positions shown in the same figure. If the code is moved, these load/store operations will be invalid as they will touch places of memory which can be occupied by any part of code or data in the destination file (figure 3.6.b). Performing these operations can result in memory corruption. In addition, trying to fit all of these different parts in three different segments can be an issue.

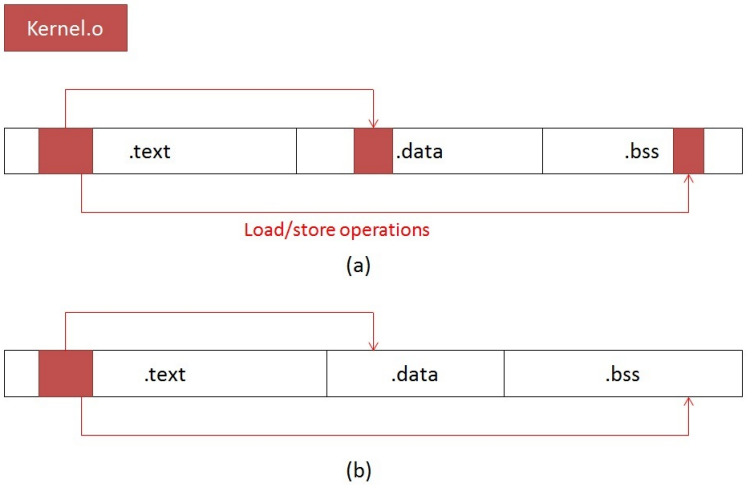


Figure 4.6.: a. Shows the kernel's object file (red) segments distributed in different segments of the final executable. b. Shows the code of the kernel moved to another executable file. Notice that the load/store addresses are not valid anymore.

For these reasons, it is necessary to have a *self contained body of code* which is movable and contains the code and the data segments associated with it, as is shown in figure 4.7.

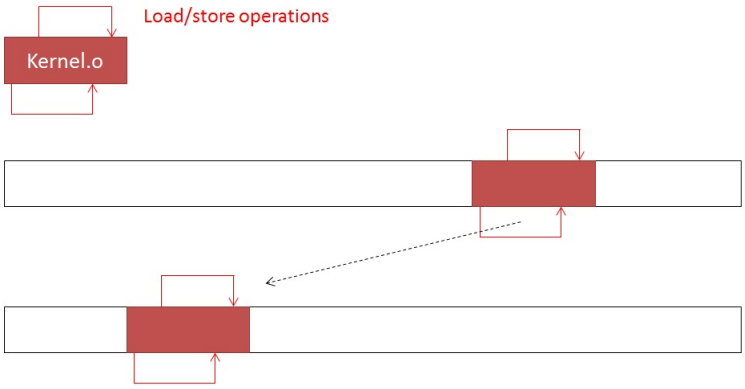


Figure 4.7.: The same object file is linked such that different segments are not spread across the executable file, making it possible for the kernel to be moved as a single body of code.

The gnu linker provides features which makes this possible to do. As was described in section (3.5), it is possible to define regions at certain positions of memory in the linker script, and instruct the linker to put specific input sections into these regions. What is important is that the linker must be able to find the intended sections among all object files which are fed to it. To make this possible, we can use the compiler [1] option "-ffunction-sections" which will instruct the compiler to preserve the section names of the function in the object file. This will help the linker to find these sections and manipulate them as instructed.

Pre-allocation of space:

Once the code is prepared and moved as a single chunk, the remaining problem is finding a place to fit the code. The issue is that different kernels have different sizes. When we copy a kernel to a core (which already contains another kernel), if the new kernel is smaller in size, there will be no problem. But when the new kernel has a larger size, if it is copied to the core, it will overwrite parts of the memory which are occupied by other sections of the program. This is illustrated in figure 4.8.

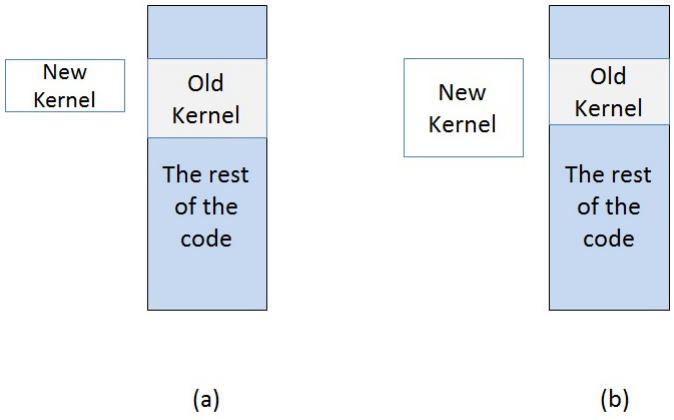


Figure 4.8.: (a) The new kernel is smaller than the old one and it fits. (b) The new kernel is larger than the old one, copying it would overwrite other parts of the program.

One solution to this problem can be pre-allocating space in memory for kernels. The size of this space should be chosen as the maximum of kernel sizes; this way, the new incoming kernels will always fit inside the area without overwriting other parts of the application. We can call this area in the memory region "kernel area".

Memory layout of worker cores

The memory layout of worker cores (the ones which will execute tasks) is illustrated in figure 4.9. The kernel area is situated at the same address on all cores, which makes it easier for BOS to invoke kernels. Storage cores will deliver kernels to this area. Furthermore, the kernels will be linked to be executed from this address (VMA, section 3.5). Please refer to appendix B.1 for the linker script of this memory layout.

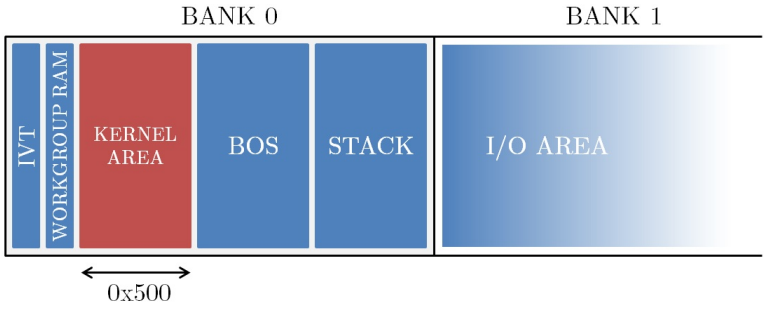


Figure 4.9.: Memory layout of ordinary cores. This illustration is not drawn to scale.

As can be seen in figure 4.9, all executables are placed in the first memory bank (BANK 0), and the rest of the memory is reserved for I/O data. Putting the code and the data

in different banks has performance benefits on the Epiphany architecture (section 3.2). To avoid internal movement of data, multiple buffering is used and the input and output pointers are situated in banks one to three. The IVT is the interrupt vector table. The WORKGROUPRAM is used by the Epiphany SDK and contains coordinate information about the core.

Memory layout of storage cores

Each storage core can store a number of kernels and supply them to other cores when they need it. A number of memory regions can be defined in the storage core's memory to host the kernels. The number of these regions is obviously limited by the available memory and the kernel sizes. The memory layout of a storage core is shown in figure 4.10.

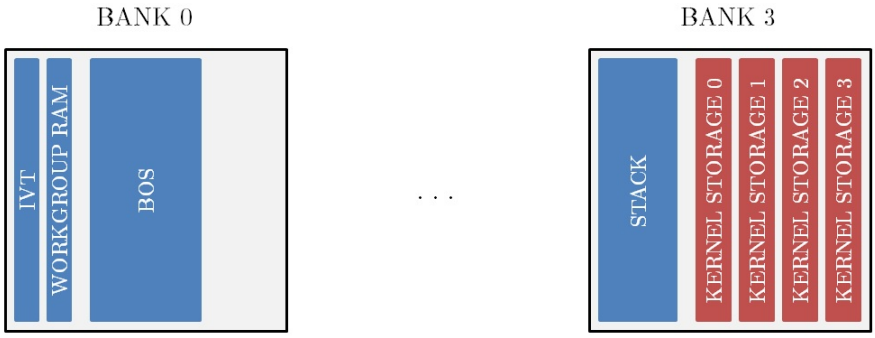


Figure 4.10.: Memory layout of storage cores. Four storage areas are allocated in bank three. The kernels will be linked to be executed from kernel area, but will be placed in one of the storage areas. The address of storage areas is provided to the C code to be used by the memcopy function. This illustration is not drawn to scale.

Using this linker technique and statically linking the kernels, has the performance benefit that kernels will be ready for execution as soon as they are copied in to a core, and the overhead of dynamic linking is avoided, since the relocation is carried out by the linker at build time.

It should also be pointed out that not all kernels will occupy a complete kernel storage bank and many of them have a smaller size. It would be inefficient to copy the entire size of the kernel storage every time a kernel needs to be delivered; instead, it is better to initiate the memcopy operation with the exact size of the kernel. The exact size of the kernel can not be predicted, as it depends on the size of the external library functions which are linked together with the kernel and also the optimization level. Fortunately, the linker can provide the total size of a section, and this value can be provided to the C code (using an external symbol) to avoid copying too many bytes of code.

It is possible to have the functionality of graph reception and kernel storage on the same core. In that case, the necessary space for holding the graph information will be allocated in other banks of the same core. Also, by having both functionalities on the same core, BOS code will become slightly larger. For that reason stack is allocated in BANK 3, since there's more space available on that memory bank.

Calling the code:

After overcoming the problems that have been mentioned so far, we still need a way of calling the kernel functions. Since at least in theory there is no limit in the number of kernels which might be moved to a core, using function names is impossible. The remaining option is function pointers. To use function pointers, the program needs to be aware of the position of the functions it is about to call in the memory. The problem now is that each kernel currently consists of more than one function (a run kernel function and a move data function). As is shown in figure 4.11, these functions can have different sizes in different kernels. All kernels will be invoked from the kernel area, and since the run kernel function will be placed in the beginning of this area, the address of it is easily available; however, the address of the move data function has a varying offset from the beginning of the kernel area, which depends on the size of the run function. Not having easy access to the address of a function can make it harder to use function pointers.

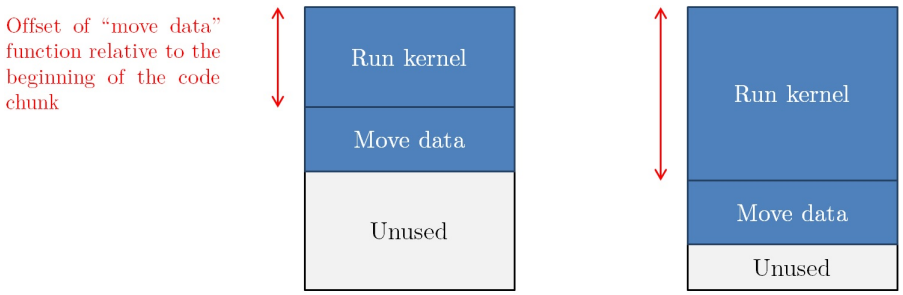


Figure 4.11.: Variation in the size of the run kernel function will affect the position of the move data function, making it more difficult to use pointers to call them.

To solve this, a wrapper can be used which calls the appropriate kernel functions. The kernel can be interfaced to BOS using a wrapper (which will be invoked using a function pointer). This wrapper takes the arguments of ordinary kernel functions plus an extra argument for choosing the function that needs to be called, which is currently either of *run kernel* or *move data* functions. The only consideration for using such wrapper is that in all of the C files in which the kernels are written, the wrapper must come as the first function. This way, when the kernel is linked and prepared as a single body of code, the wrapper will be placed in the beginning of the section. This is because the linker will not shuffle sections around [13].

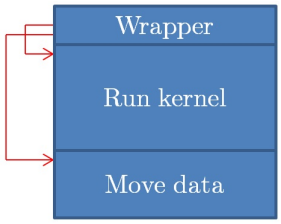


Figure 4.12.: A wrapper can be inserted on top of every kernel C file to call other functions.

4.5. Further optimization

One optimization which was not implemented due to time limitations, is double buffering of kernels. The kernel area on the core can be divided into two separate areas, and while one kernel is being executed, the next one can be copied into the memory simultaneously. This way, once a task is finished, the next one can start immediately; however, on the Epiphany architecture, having a piece of code executing and having an external agent copying into the same memory bank where the code is executed from, can hit the performance. On Epiphany, in every instruction fetch 64 bits of instruction are fetched on a single clock cycle into the instruction sequencer [3], but the length of instructions is shorter than 64 bits which means there is no need to fetch an instruction on every clock cycle. It would be hard to predict the exact amount of performance loss due to multiple agents accessing the same bank since it depends on the code which is being executed, but in the tests that have been carried out the performance loss was negligible.

4.6. Benchmarking

To benchmark the presented solution, an artificial parallel application with two configurations was created. The purpose is to compare the overhead of moving code with that of moving data. The two applications consist of a set of tasks coming in series, but have a different graph design to force either of code movement or data movement.

The application consists of five tasks coming in sequence. In the first configuration, whose purpose is to measure the overhead of data movement, the graph is constructed using ordinary BOS graph elements: five tasks coming in sequence with a barrier between every two tasks. This is shown in figure 4.13.a. Each task uses a different kernel and there are five different kernels in total. The kernels are mapped to different cores(figure 4.13.b). This forces the tasks to be executed on different cores, and thus, the data to be moved between the cores.

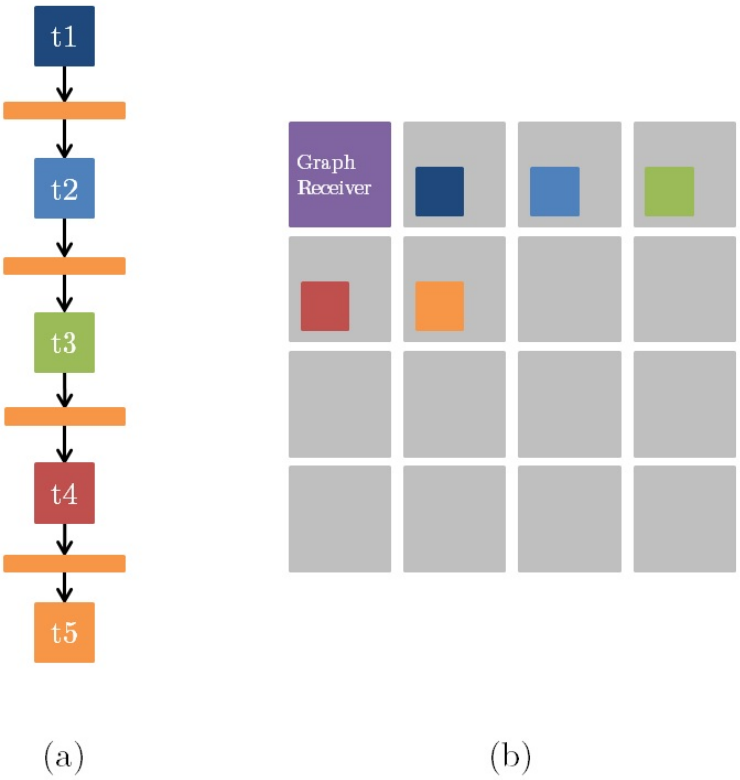


Figure 4.13.: A series of tasks mapped to different cores, to measure communication overhead of moving data.

In the second configuration, the aim is to measure the overhead of code movement. To do this, the same tasks with the same kernels are used, but this time the tasks t2 to t5 are connected using the sequential task construct (figure figure 4.14).a). The first task is mapped to one core, and the sequential tasks (t2 to t5) are mapped to another core (as shown in figure 4.14).b. This core initially supports the kernel of the first task of the sequential chain. After this task is executed, the rest of the kernels will be fetched and executed in order. Also, as can be seen in the figure, the storage core stores all kernels required in the sequential chain, including the first one (the first kernel will be post-fetched after all tasks are executed).

Please note that the position of storage core is next to the core which executes sequential tasks. This is to ensure that the measurements are carried under identical conditions: the measured communication time is between two neighboring cores in both cases (data move and code move). Also, at the time that these communications are going on, no other communication is active anywhere in the network.

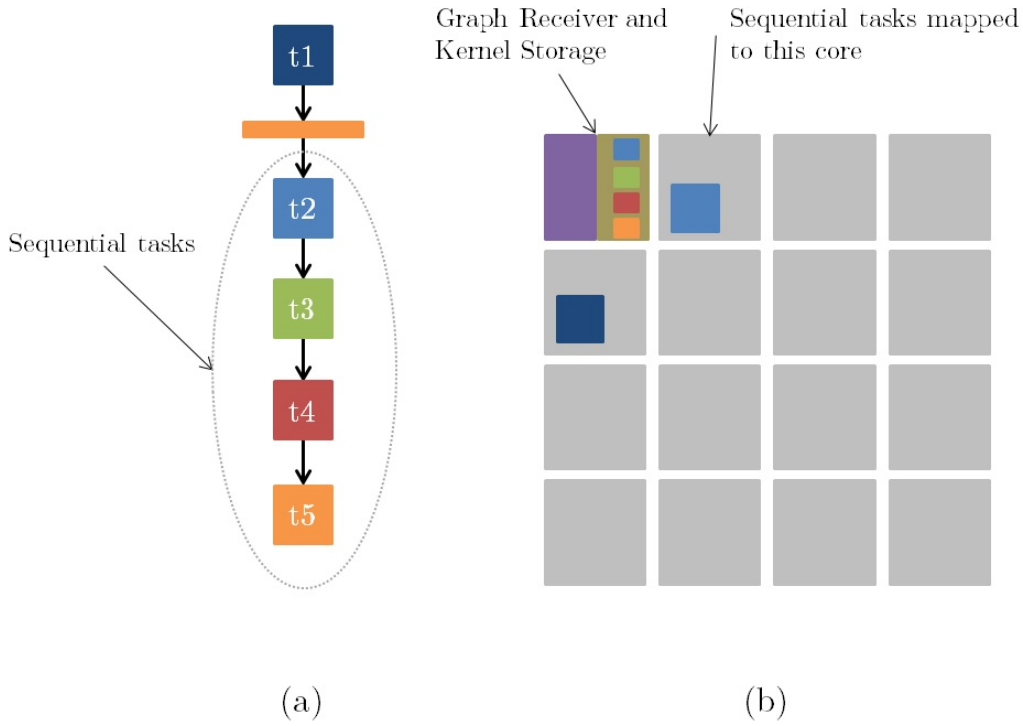


Figure 4.14.: A graph exploiting new sequential task feature, designed to measure the overhead of moving code.

The question that we're trying to answer here is: how much larger should the data size be so that it becomes beneficial to move the code instead. To simplify the comparison, a fixed value was chosen as the size of the moved code (the memcopy function was instructed to move a fixed of code, regardless of actual size of the kernel), but different values for data size were examined. Both graphs share exactly the same kernels. The application does not have any particular purpose and the kernels only touch data elements and add a value to them in a loop.

4.7. Results and analysis

In this section, performance measurements of the benchmarking applications are presented. A tool developed at Ericsson was used to interpret trace outputs and graphically demonstrate the events and the timings of the application. Figure 4.15 and 4.16 contain these illustrations. In these plots, the horizontal axis represents the time, and each row represents the events occurring on each core. The pink bar indicates the execution of a kernel, the blue bar indicates the search for next task's kernel support among other cores (carried out by BOS), the cyan bar represents the data movement and finally, the red bar represents copying a kernel kernel. Also, the brown colored bar is the graph conversion time on the graph receiver core.

In figure 4.15, the events from the first configuration of the application (move data) are demonstrated. In this application, the kernels are mapped on different cores, and

in this way, the data is forced to travel between the cores. As can be seen, each core's time-line shows execution of a kernel, searching among other cores for the next task's kernel support, and moving the output data.

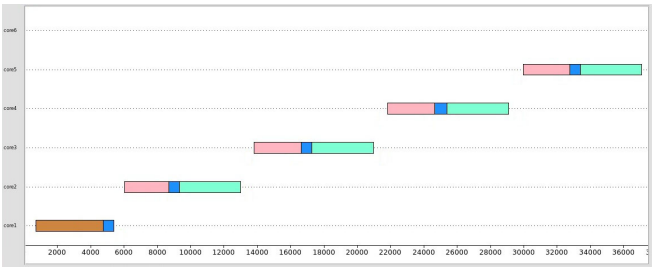


Figure 4.15.: The events occuring in the hardware in the first configuration (executing each task on a different core and moving data between the cores). The color pink is task execution, cyan is the time for moving data and dark blue is the platform overhead to find a core for the next task.

In figure 4.16, the events from the second configuration (move code) are illustrated. In this configuration, the tasks t2-t5 will be executed on core 3. As can be seen, after the graph is submitted and the conversion is carried out on core 1, the first task is executed on core 6 and then its output data is moved to core 2, which will proceed with the sequential chain. After that, the task t2 is started on core 2 (the core to which sequential tasks are mapped t), and after its execution, the kernel for the next task, k3, is fetched (the red bar). The gap between the execution (pink) and kernel fetch (red) is the platform overhead of communicating to the graph receiver core to get the graph information. Also note that the red colored bars are in pairs; for every code movement that happens one bar appears on the time-line of the storage core and another bar appears on the time line of the requester core. The later is slightly longer because the starting point of the event is when the message is sent, while on the storage cores, the starting point of the event is when the message is received on the storage core (the delay from inter-core communication causes this difference).

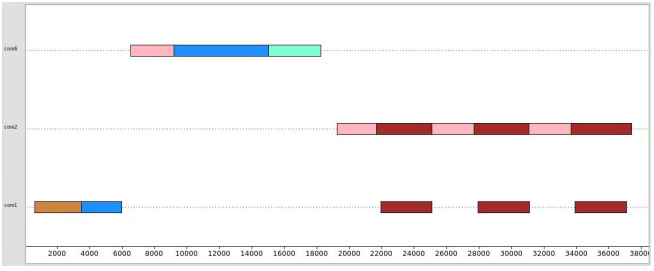


Figure 4.16.: The events of the second configuration (core 2 executing tasks sequentially while the code for different kernels is being delivered to it). The red bars represent the time it takes to move code between the cores. For each kernel delivery, two red bars appear in the graph: one in the sender core, and one in the receiver.

As was explained, in the first configuration (data move) different data sizes were examined. The measurements of communication time for task t3 in the two different applications is presented in tables 4.17 and 4.18.

	Data size (Bytes)	Communication time (Cycles)
1	200	543
2	400	1043
3	600	1543
4	800	2043
5	1000	2543
6	1200	3044
7	1400	3544
8	1600	4044
9	1800	4544
10	200	5044

Figure 4.17.: Communication time in cycles for moving data, for task t3.

Code size (Bytes)	Communication time (Cycles)
1200	3043

Figure 4.18.: Communication time in cycles for moving code for task t3.

It should be mentioned that the all measurements are from the sender core's point of view, and the values presented in the tables do not include any extra overheads, except for the cycles it takes to read the timer values (which is consistent in all measurements).

In the diagram presented in figure 4.19, these results are compared. The horizontal axis represents the data size, the red curve represents the communication overhead of data movement (in cycles), and the blue curve is the communication overhead of code movement for a fixed size.

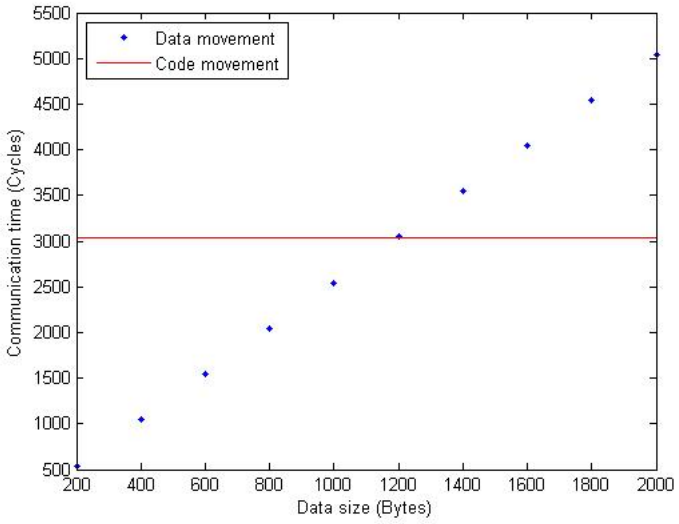


Figure 4.19.: Comparison of code movement and data movement communication overheads. The red line represents communication time for code movement for a fixed size, and the blue dots represent the communication time for data movement for the sizes indicated on the horizontal axis.

After comparing figures 4.15 and 4.16, we can observe that the platform overhead for starting a new task is significantly reduced when we execute tasks sequentially. The reason for that is when we execute tasks sequentially, the core is already aware of the location of the storage core (this information is currently hard-coded, but it can be automatically generated by a future code generator). This means that the BOS overhead of searching among cores, which normally happens after a task is finished, is eliminated. However, we can not consider this as an advantage for the proposed method, as it is BOS specific.

We can see in the measured results, which can be found in tables 4.17, and 4.18, that there's no significant difference in the communication time of moving code and moving data given that the sizes are the same. Furthermore, there is no extra overhead of dynamic linking and relocation after code movement. Also, the code size for handling sequential tasks is quite small. This can bring us to the conclusion that moving code has the same cost as moving data for similar sizes, with no extra overhead. Thus, when the data size is greater than code size, the movement of code is beneficial; however, there can be a few considerations on implementing this method:

1. Moving code in parts of the parallel application graph with fork/join construct can be quite complicated. This fact along with the condition that the code size must be smaller than the data size assigned to a specific task, will narrow the set of application candidates for this optimization. It should also be noted that in case of sequential tasks, if the code size of the tasks are small enough that they would fit on the same core, they should be statically mapped to the core; this way, there would be no need for moving the kernels.

2. The implementation of this method, requires deeper knowledge of the compiler toolchain, in particular the linker, and also the hardware. The complications and hassles of this method, can be abstract away from the application designer by using a code generator, and providing a front end which abstracts away all of the implementation details.

3. Also, his method would require using some of the cores of the hardware for the purpose of storing kernels. This can possibly limit the use of these cores for other purposes, as they will have memory limitations.

5

Conclusions and future work

The question that we tried to answer in this thesis work was whether or not it can be useful to move the code instead of data in case it is smaller in size, and how difficult this can be. A new method has been invented using the features of the gnu linker which have been around for several years, and this method was successfully integrated into BOS. Furthermore, we have created an artificial application in order to benchmark the implemented solution.

The results from the test show that there is no difference in the communication overhead of moving code and moving data, given that the sizes are the same. Also, the implemented method does not introduce any extra overheads of dynamic linking or relocation for code movements, since the relocation is done at build time by the linker and the code is ready for execution once it is copied in. Furthermore, the added code size to enable this functionality, and its execution time is very small and doesn't occupy much memory.

Having no extra overheads for moving code, and having the same communication time for the same size, brings us to the conclusion that as soon as the code size is smaller than the data size, it is beneficial to move the code.

5.1. Future work

The main purpose of developing BOS has been to be able to support dynamically altering parallel applications, by being able to accept applications with different graph description. This can be further complemented by dynamically changing the supported kernels on the hardware, and a method very similar to the one developed in this thesis work can be used to achieve this.

In a situation where we have a large number of tasks coming in sequence with small code sizes, a possible optimization is double buffering for kernels. The successor kernel can be loaded to the core, while it is executing a task.

Implementing double buffering for kernels could have an additional consideration: If this method is implemented, it would be better to put the two buffers right next to each other, because this way the larger kernels can occupy the space of both buffers; however, on the Epiphany architecture, having both buffers in the same bank can slightly decrease the performance of the execution, because while a kernel is being executed another one would be being loaded into another buffer in the same bank. There is no way of predicting the exact amount of performance loss, although most probably it won't be too high. The reason for that is that on Epiphany, every instruction fetch will load 64 bits of instruction into the instruction sequencer, but the length of instructions are shorter than 64 bits. This means that there will be two agents competing to access the same bank, but the instruction fetch is not as demanding as the other one.

As is discussed in appendix B, the proposed method currently has a limitation regarding the use of static libraries in the kernel code. The current method would require the programmer to follow a specific naming convention for kernel's function names, which would be used by the linker to recognize them and manipulate them in our desired way. This method is not proven to work when code from static libraries is included in the kernels, since they obviously do not follow the same naming convention.

Finally, as the implementation of this method can be difficult and tedious, automatic code generation in order to abstract away all of the low level implementation details can be highly advantageous.

A

Linker scripts

The job of the linker is to merge the input object files and produce an executable file. In order to do so, the linker will map the input sections of the object files into output sections. The programmer can control this process using linker scripts. A linker script, among other things, will tell the linker how to control the memory layout of the output file, and how to map the input sections into the output file [13]. The linker script for the gnu linker must be written in the *command language* and saved with *ldf* extension.

The main body of a linker script consists of two commands: the MEMORY command and the SECTIONS command. The only mandatory command is SECTIONS, using which one can specify how the input sections are mapped to the output sections. If the SECTIONS command is used alone, all addresses need to be specified explicitly; however, this can be simplified using the MEMORY command. Using the MEMORY command, one can define regions in the memory. The region names defined in the MEMORY command can then be used in the SECTIONS command.

MEMORY command:

The linker's default configuration permits allocation of all available memory. This can be overwritten using the MEMORY command (figure A.1). MEMORY command can be used to describe memory regions of the target hardware's memory. This description can be used to tell the linker which regions must be used and which ones must be avoided. The region names, can be also used inside the SECTIONS command to assign input sections to specific memory regions. As has been mentioned before, the MEMORY command is optional in the linker script [13].

```
MEMORY
  {
    name [(attr)] : ORIGIN = origin, LENGTH = len
    ...
  }
```

Figure A.1.: The MEMORY command can be used to define memory regions. Each region will be assigned a name, and it is defined using its starting point in memory and its length.

Optionally, each memory region definition can have a list of attributes which will tell

the linker whether it can use that region for input sections which are not explicitly mapped to an output section. These attributes can be found in table A.2:

Attribute	Meaning
R	Read-only section
W	Read/write section
X	Executable section
A	Allocatable section
I	Initialized section
L	Same as I
!	Invert the sense of the attributes which follow

Figure A.2.: Attributes used to describe memory regions.

SECTIONS command:

The SECTION command is the only mandatory command which must be present in the linker script. It is used to manipulate the segment addresses in the final object file. The SECTIONS command tells the linker how to map input sections into output sections, and how to place the output sections in memory [13].

```
SECTIONS
{
   sections-command
   sections-command

}
```

Figure A.3.

Each *sections-command* can be either of the following:

1. an ENTRY command
2. a symbol assignment
3. an output section description
4. an overlay description

The ENTRY command and overlay description will not concern us in this thesis work, for more details about them please refer to [13]. The output section description, which is describe below, is used to manipulate program sections.

Output section description

The output section description is used to tell the linker how to map input sections into output sections. The body of output section description is shown in figure A.4. Output section description will describe:

1. Which input section will be mapped to which output section
2. Where will the output section be executed from (Virtual memory address, described later in this section)
3. Where will the output section initially be loaded (Load memory address, described later in this section)

```
section [address] [(type)] :
  [AT(lma)]
  [ALIGN(section_align) | ALIGN_WITH_INPUT]
  [SUBALIGN(subsection_align)]
  [constraint]
  {
    output-section-command
    output-section-command
    ...
  } [>region] [AT>lma_region] [:phdr :phdr ...]
    [=fillexp]
```

Figure A.4.: The full description of an output section.

Each output section description can take several attributes, most of which are optional. Among them are VMA and LMA. VMA stands for Virtual Memory Address, and it is the address at which the code is assumed to be situated during the execution time. In other words, it is the address the code is resolved to be executed from. LMA or the Load Memory Address, is the address at which the section will be loaded initially. LMA is an optional attribute, and if omitted, it will default to the given VMA. In most cases VMA and LMA are equal, but there are situations in which it is useful to have different addresses for them:

1. Most processors used in embedded systems have von Neuman architecture, which means a single address space will address the read-only programmable flash, and the program memory (RAM). If the program uses any initialized data (static variables or global variables), it will expect it to be in the program memory when it starts. But it's not possible to initially put initialized data in program memory, since it will lose its data at the absence of power. It is also impossible to put the data in flash, as it is read-only memory. In such cases, the data is defined to be in a section whose VMA is in RAM, but its LMA is in flash. This way, the linker would resolve the addresses as if it would be accessed in RAM, but it will load

it in flash. When the chip is powered, some anonymous function will have the responsibility of moving the data to RAM before the control is transferred to the main() function. These anonymous functions are usually put in the .init section and are run before the control is transfered to main.

2. In another class of scenarios, again in embedded systems, it is common with many manufacturers to produce chips which have a faster RAM but a slower flash. In such cases, if the code is put into ROM, the processor might not be able to run at its highest performance as many stalls will be inserted into pipeline while the instructions are being fetched (due to the slow ROM). In such cases, usually the critical parts of the code linked as if they would be executed from RAM (VMA in RAM) but they are loaded into the flash initially (LMA in flash). A run-time loader is then expected to copy in these functions before they are called.

It should be noted that it is possible for several sections to share the same VMA. The linker will not generate any errors or warnings for such case. If overlapping VMAs are used, a run-time function must copy the code into place before it is executed. It is practically impossible to have more than one section mapped to the same LMA. The linker will generate an error if this happens. Several input sections can be mapped to an output section. The input sections can be file names followed by a list of section names.

In each output section description, the *section* field is the name of the output section. The VMA can be specified either in the '[address]' field or the '[>region]' field. If '[address]' is used, an explicit address has to be provided, and if the '[>region]' field is used, the name of a region defined inside the body of the MEMORY command can be provided. Specifying LMA is not mandatory, to specify the it, an explicit address can be provided in the '[AT(lma)]' field, or, the name of a region can be provided in the '[AT >lma_region]'. Also, we can indicate the input sections which are supposed to be put at the intended output section. *output-section-command* is used for this purpose, and the input section names can be provided specifically or using wild cards. For more information please refer to [13].

A.1. Accessing linker defined symbols from code

For several reasons it is useful to be able to use linker defined symbols in the code. In this section it is shown how to use symbols defined in the linker script in a high level language like C.

If we define a symbol in the liker script and intend to use it in a high level language (in our case, C), the value of the symbol will be unknown to the compiler, and must be resolved at later stages by the linker. For that reason, such symbols must be defined as external symbols in the C code. The produced object file will contain this symbol and the value will be resolved by the linker. Another important point is about the naming of the symbols. Compilers transform the symbol names before storing them into the symbol table. This is different from compiler to compiler, but the C compiler used

in the Epiphany's tool-chain, inserts an underscore before every symbol name. This means that when the object code reaches the linking stage, the symbol entry will carry one extra underscore prefix. For that reason, the symbol which is defined in the linker script should be prefixed with one extra underscore so that the symbol names match at linking stage. The definition and usage of these are shown in figures A.6 and A.5.

```
_foo = 1000;
```

Figure A.5.: Definition of a symbol in the linker script. Notice the extra underscore.

```
extern int foo;

/* Accessing the value */
void * value = &foo;
```

Figure A.6.: Definition of a symbol in C code.

The reason behind the C syntax is explained below: when a symbol is declared at a high level language, the compiler will reserve enough space in the program memory to hold the symbol's value, and it will create a new entry in the program's symbol table which will hold the symbol's address. Each entry in the symbol table has a value and an address. The address in the memory will not be known by the compiler and will be resolved by the linker. For ordinary variables, when the variable is accessed, the compiler will generate a code which will get that symbols address, and access the value of the symbol at that address in the memory. The difference with linker defined symbols is that for them, the value of the symbol is stored in the address field of the symbol table. This is the reason that in the C code the '&' operator is used. It is illegal to directly access the value of a linker defined symbols. The type of the 'value' variable is not important. Any type can be used and the value can later be type-cast to other types.

B

Implementation

The implementation details of the improvised method are discussed in this section. It is discussed how the kernel codeis manipulated into the storage areas of the storage cores, in a way which makes them accessible and executable by other cores.

As has been discussed before, it is important to keep the kernel code in a single chunk. In order to achieve this, it was decided to put the code for each kernel in a single C file, and use a specific naming convention for the kernel functions. This will make it easier later to use wild cards in the linker script to identify them as input sections. Also, the wrapper must come as the first function in the C file, so that it ends up in the beginning of the kernel area. After this function, the run kernel and the move data functions can come. The linker will not shuffle the functions around [13].

The naming convention for the function names is that each function name must end with the "__XX__KERNEL" string, with "XX" being the kernel number. This way, we can easily refer to a specific kernel using the "*__XX__KERNEL" wild card, or, to all of the kernels in general using the "*__KERNEL" wild card. Generally, we will need to refer to kernels to put them in the kernel area for worker cores. But for the kernels which are to be put in the storage area, we will be able to filter them in using a wild card with their specific number.

Furthermore, to compile the kernel code, the "-ffunction-sections" option must be used. This will instruct the compiler to place each function into its own section in the output file. The name of the function or the name of the data item determines the section's name in the output file [1]. This way, the function name that we have chosen will turn into a section name, which is easy to differentiate later by the linker.

It is important to note that this method requires us to follow a specific naming convention for function names, and enable the compiler option mentioned above. Unfortunately, this will limit us to using only functions whose source codes are available and therefore static libraries can not be used in the kernel code which is to be moved between Epiphany cores.

B.1. Memory layout of worker cores

It has been mentioned in chapter 4 that on worker cores, one area must be dedicated to kernels and be excluded from the rest of the memory regions. To do this, in the linker

script we can first use the MEMORY command to define the memory regions that we need and then in the SECTIONS command, use these regions as the destination for the program sections. In figure B.1, the MEMORY command for worker cores is shown. This is mostly similar to most of default linker scripts, except that the KERNEL_AREA is inserted before BANK0. Note that the starting point of BANK0 and INTERNAL_RAM is shifted with a value equal to the length of KERNEL_AREA. This will *exclude* the kernel area from the rest of the regions. The length of the KERNEL_AREA is chosen to be 500 hex equal to 1280 Bytes.

```
/* run time lib and crt0 */
IVT_RAM (WXAI)          : ORIGIN = 0, LENGTH = 0x28

/* user program, continuous placement */
WORKGROUP_RAM (WXAI)    : ORIGIN = LENGTH(IVT_RAM),
   LENGTH = 0x28

/*Kernel Area: In this area only kernels can be loaded,
   and it is excluded from other regions*/

KERNEL_AREA (WXAI)      : ORIGIN = LENGTH(IVT_RAM)
   + LENGTH(WORKGROUP_RAM) , LENGTH = 0x500

/* user program, continuous placement */
INTERNAL_RAM (WXAI)     : ORIGIN = LENGTH(IVT_RAM)
   + LENGTH(WORKGROUP_RAM) + LENGTH (KERNEL_AREA),
     LENGTH = 32K - LENGTH(IVT_RAM)
   - LENGTH(WORKGROUP_RAM) - LENGTH (KERNEL_AREA)

/* user program, per bank usage */
BANK0_SRAM (WXAI)       : ORIGIN = LENGTH(IVT_RAM)
   + LENGTH(WORKGROUP_RAM) +  LENGTH (KERNEL_AREA),
     LENGTH = 8K  - LENGTH(IVT_RAM)
   - LENGTH(WORKGROUP_RAM) - LENGTH (KERNEL_AREA)
BANK1_SRAM (WXAI)       : ORIGIN = 0x2000, LENGTH = 8K
BANK2_SRAM (WXAI)       : ORIGIN = 0x4000, LENGTH = 8K
BANK3_SRAM (WXAI)       : ORIGIN = 0x6000, LENGTH = 8K
```

Figure B.1.: Linker commands defining the memory regions of ordinary

Different sections of the code must be manipulated to be put into their respective regions. What concerns us, is the kernel code. This is the same for all of the cores. It has been described in appendix A how we can manipulate the input and output sections using the SECTIONS command. In figure B.2 it is shown how we can tell the linker to put the kernel code in the kernel area. The wild card (*.o(*__KERNEL)) refers to all sections whose names end with "__KERNEL" in all object files. ".kernel_area" is the name of this section. "ORIGIN(IVT_RAM) + LENGTH(IVT_RAM)

+ LENGTH(WORKGROUP_RAM)" is the VMA (explained in section 3.6) for this section, which matches the "KERNEL_AREA" region defined in the body of the MEMORY command. In this case, VMA and LMA are the same, and the contents of this section will be loaded to the same address. For most cores, kernel area would contain the kernel(s) that the core supports, And for the cores which support sequential tasks, the kernel area contains the first kernel of the sequential branch (there is no need for alternation in the linker script for sequential cores). Also, the address of the starting point of the kernel area must be provided to the C code, so that the cores executing sequential tasks can use this address for calling the wrapper. The starting point is saved in a symbol called __KERNEL_START. This value depends on the lengths of the previous regions.

```
.kernel_area ORIGIN(IVT_RAM) + LENGTH(IVT_RAM)
    + LENGTH(WORKGROUP_RAM): {*.o(*__KERNEL)} > KERNEL_AREA

__KERNEL_START = ORIGIN(IVT_RAM) + LENGTH(IVT_RAM)
    + LENGTH(WORKGROUP_RAM);
```

Figure B.2.: sections command in linker script, manipulating where the kernel code should be placed

B.2. Memory Layout of Storage Cores

The storage core(s) will contain the kernel code of several kernels. The number of kernels which can be fit into them depends on the available memory and the kernel sizes. A region in which kernels are stored are called KERNEL_STORAGE. The MEMORY command for the storage cores is shown in figure B.3. BANK3 is used to store kernels and four kernel storage regions are defined, each having a size of 500 hex (1280 Bytes). The rest of this bank is used for the stack. BANK3 is not defined anymore and this area is excluded from the INTERNAL_RAM region. The kernel area is not defined anymore, since storage cores will never run a task.

```
/* run time lib and crt0 */
IVT_RAM (WXAI) : ORIGIN = 0, LENGTH = 0x28

/* user program, continuous placement */
WORKGROUP_RAM (WXAI)      : ORIGIN = LENGTH(IVT_RAM),
   LENGTH = 0x28

/* user program, continuous placement */
INTERNAL_RAM (WXAI)       : ORIGIN = LENGTH(IVT_RAM)
   + LENGTH(WORKGROUP_RAM), LENGTH = 24K - LENGTH(IVT_RAM)
   - LENGTH(WORKGROUP_RAM)

/* user program, per bank usage */
BANK0_SRAM (WXAI)         : ORIGIN = LENGTH(IVT_RAM)
   + LENGTH(WORKGROUP_RAM), LENGTH = 8K
   - LENGTH(IVT_RAM) - LENGTH(WORKGROUP_RAM)
BANK1_SRAM (WXAI)         : ORIGIN = 0x2000, LENGTH = 8K
BANK2_SRAM (WXAI)         : ORIGIN = 0x4000, LENGTH = 8K
/* Kernel storage areas */
STACK_AREA (WXAI)         : ORIGIN = 0x6000, LENGTH = 0xC00
KERNEL_STORAGE0 (WXAI)    : ORIGIN = 0x6C00, LENGTH = 0x500
KERNEL_STORAGE1 (WXAI)    : ORIGIN = 0x7100, LENGTH = 0x500
KERNEL_STORAGE2 (WXAI)    : ORIGIN = 0x7600, LENGTH = 0x500
KERNEL_STORAGE3 (WXAI)    : ORIGIN = 0x7B00, LENGTH = 0x500
```

Figure B.3.: Linker script commands defining the memory regions of storage cores

In the SECTIONS command, we can instruct the linker on how to link and position the output sections. What is important is that the kernel codes will not be executed on this core, instead, they will be executed on other cores, from a position which is known beforehand: kernel area. The linker must link and resolve each kernel's code as if it would be executed from the kernel area, but it should load them into one of the storage cores. The kernel area region is not defined in the linker script for storage cores, but its preceding regions are defined and therefore we can calculate the starting address of this region based on those. Output section description for kernel storage can be found in figure B.4. The VMA which will be provided to the command is the starting point of the kernel area. A symbol called "__KERNEL_START" is defined and its value is calculated based on the lengths of the previous sections. This symbol is used to provide the value of the VMA, and, it will be imported as an external symbol into the C code, to help the calculation of destination address for kernel movement. Each kernel must be copied to the kernel area of another core. The requester core will send its own address in the request, and this constant provides the offset of kernel area.

The value of the LMA of each storage area (indicated with "$AT > ...$"), is its respective storage region, defined in the body of the MEMORY command. The kernels will be initially loaded to these addresses. When this core is requested to supply a

kernel, a memcopy command will be initiated. This command will need to be provided with the address at which the kernel resides. For this purpose, one symbol (__KERNEL_STORAGEX_START) is defined for each of the kernel storage regions and is set to starting point of that region.

Another feature that the gnu linker provides is calculating the total size of each of the sections. This value depends on the optimization level and can only be calculated by the linker after the linking is done. The size of each section is calculated after it is filled, using the "SIZEOF" command. The size values are put in "__STORAGE_SIZE_X" symbols to be used by the C code. As a further optimization, this is value is later used in the memcopy operation to copy just enough number of bytes.

```
__KERNEL_START = ORIGIN(IVT_RAM) + LENGTH(IVT_RAM)
    + LENGTH(WORKGROUP_RAM) ;

.kernel_storage0 __KERNEL_START : {*.o(*__05__KERNEL)}
    AT>KERNEL_STORAGE0
__STORAGE_SIZE_0 = SIZEOF(.kernel_storage0);

.kernel_storage1 __KERNEL_START : {*.o(*__02__KERNEL)}
    AT>KERNEL_STORAGE1
__STORAGE_SIZE_1 = SIZEOF(.kernel_storage1);

.kernel_storage2 __KERNEL_START : {*.o(*__03__KERNEL)}
    AT>KERNEL_STORAGE2
__STORAGE_SIZE_2 = SIZEOF(.kernel_storage2);

.kernel_storage3 __KERNEL_START : {*.o(*__04__KERNEL)}
    AT>KERNEL_STORAGE3
__STORAGE_SIZE_3 = SIZEOF(.kernel_storage3);

__KERNEL_STORAGE0_START = ORIGIN(KERNEL_STORAGE0);
__KERNEL_STORAGE1_START = ORIGIN(KERNEL_STORAGE1);
__KERNEL_STORAGE2_START = ORIGIN(KERNEL_STORAGE2);
__KERNEL_STORAGE3_START = ORIGIN(KERNEL_STORAGE3);
```

Figure B.4.: Linker script commands which manipulate placement of kernels in the storage area. tell the linker resolution address (VMA) and the loading address (LMA)

Bibliography

[1] *GCC online documentation.*

[2] *Parallella Reference Manual*, (REV 13.11.25).

[3] Adapteva. *Epiphany Architecture Reference Manual*, REV 14.03.11, 2014.

[4] Michael Barr and Anthony Massa. *Programming embedded systems: with C and GNU development tools.* " O'Reilly Media, Inc.", 2006.

[5] Ciprian Docan, Manish Parashar, Julian Cummings, and Scott Klasky. Moving the code to the data-dynamic code deployment using activespaces. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 758–769. IEEE, 2011.

[6] Michael Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.

[7] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach.* Elsevier, 2012.

[8] John R. Levine. *Linkers and Loaders.* The Morgan Kaufmann Series in Software Engineering and Programming. Morgan Kaufmann, 1 edition, 1999.

[9] Kwan-Liu Ma. In situ visualization at extreme scale: Challenges and opportunities. *Computer Graphics and Applications, IEEE*, 29(6):14–19, 2009.

[10] Arch Robison Michael McCool, James Reinders. *Structured Parallel Programming: Patterns for Efficient Computation.* Morgan Kaufmann, 1 edition, 2008.

[11] Peter Pacheco. *An Introduction to Parallel Programming.* Morgan Kaufmann, 1 edition, 2011.

[12] Ian Lance Taylor. A new elf linker. In *GCC Developers Summit*, page 129, 2008.

[13] Steve Chamberlain & Ian Lance Taylor. *The GNU linker.*

[14] Hongfeng Yu, Chaoli Wang, Ray W Grout, Jacqueline H Chen, and Kwan-Liu Ma. In situ visualization for large-scale combustion simulations. *IEEE Computer Graphics and Applications*, 30(3):45–57, 2010.

Farzad Farahaninia

PO Box 823, SE-301 18 Halmstad
Phone: +35 46 16 71 00
E-mail: registrator@hh.se
www.hh.se