CHALMERS

Adaptive core assignment for Adapteva Epiphany

Master of Science Thesis in Embedded Electronic System Design

Erik Alveflo

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden 2015

Adaptive core assignment for Adapteva Epiphany
Erik Alveflo,

Examiner: Per Larsson-Edefors

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden 2015

Adaptive core assignment for Adapteva Epiphany

Erik Alveflo
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

The number of cores in many-core processors is ever increasing, and so is the number of defects due to manufacturing variations and wear-out mechanisms. Sacrificing cores for increased longevity and manufacturing yield is cheap when thousands of cores are available. We propose two systems for tolerating faulty cores through software reconfiguration. The systems are based on remapping the user application when faults occur. Tasks are moved from faulty cores to reserved spare cores. Both systems were developed using a prototype based approach. One is inspired by the Epiphany Software Development Kit while the other utilizes simulated annealing to optimize distances between communicating tasks. Simulated annealing can automatically map applications while adhering to design constraints, but it can take 40 minutes to map an application with 4000 cores using the Parallella board's ARM processor.

Acknowledgements

I would like to thank my supervisor at Chalmers, Sally McKee, and my supervisor at Ericsson, Peter Brauer. I would also like to thank Lars Svensson and Per Larsson-Edefors at Chalmers, and Roland Carlsson M, David Engdal, Martin Lundqvist, Aare Mällo, Leif Spaander, and Anders Wass at Ericsson. Thank you for your great support, your feedback and everything else that made this thesis possible.

Erik Alveflo, Göteborg, 2015

This work was carried out for Ericsson, on their site at Lindholmen, Göteborg Sweden. For more information about the company, please visit their website
`http://www.ericsson.com`

Contents

1

Introduction

The number of cores in many-core processor chips continue to increase, and so does the risk of manufacturing defects. However, the impact of a faulty core decreases with the number of available cores. If a single core out of 4000 is faulty the large majority of cores can still be used. Discarding a partially faulty chip is not a cost efficient solution. An efficient method for tolerating core failures must be found. This becomes increasingly important as the number of cores increases.

Complex computer systems can comprise of hundreds of cores. When much of a product's functionality is controlled by a single chip, the importance of tolerating failures in that chip grows.

To complicate matters further, chip manufacturers may allow faulty cores in order to increase their yield, leaving developers to deal with their faulty cores as they deem necessary. When designing a software system, these concerns should be addressed from the start via a clear strategy. If available cores differ from chip to chip, the developer cannot make assumptions about which cores to trust or not. The software must be able to run across different core configurations.

1.1 Purpose and goal

The purpose of this master's thesis is to investigate dynamic core assignment and how it can be used to implement a remapping system that can handle faulty cores in a many-core processor. Dynamic core assignment can tolerate core faults by remapping the application to other cores when a fault is detected. The fault triggers a remap which reassigns each task to non-faulty cores. The detected fault is tolerated by avoiding the core in which it occurred.

Remapping an application will change its core configuration which will likely lower its performance because the distance between cores affects communication latencies and congestion. We therefore seek an automated approach to mapping parallel applications

to chips with faulty cores in ways that do not hurt performance and may even help it.

1.2 Method

In this thesis remapping is investigated and remapping systems are proposed, implemented and evaluated. We strive to:

Investigate remapping as a fault tolerance method. We first investigate the strengths and weaknesses of remapping. We then study the application requirements before proposing a remapping system.

Propose fault tolerant remapping systems. Understanding the system as a whole and the requirements it poses on the application is necessary when implementing it. We therefore propose a system and evaluate it before proceeding to the implementation.

Implement the proposed systems. A proof-of-concept implementation of the proposed system will allow us to measure its performance impact. It will also make it easier to understand the system requirements and pitfalls in the proposed method.

Estimate performance impact of remapping. How will the solutions affect the performance of the application?

1.3 Limitations

Faulty cores can be detected by running a built-in self test, by running test software, or by other means. However, how faults are detected is beyond the scope of this thesis. Our solutions rely on external error detection and fault analysis. Furthermore, only permanent faults are targeted: intermittent and transient faults are ignored.

Run-time reconfiguration is beyond the scope of this thesis. Applications are remapped while off-line. Therefore, the amount of parallelization cannot depend on input data.

Run-time recovery is beyond the scope of this thesis. Application state is discarded if faults occur during run-time. However, the application can implement recovery alongside the proposed remapping system.

Applications are not allowed to change configuration during run-time. For the purpose of this thesis the generated assignment is static during run-time; it is only allowed to change at startup.

We use the Adapteva's 64-core Epiphany processor as a reference platform but the results should apply to similar hardware. The Epiphany processor is well suited for this thesis, as it does not try to handle hardware faults: it features no built-in redundancy or error checking. The application software is free to deal with faulty cores in any way possible.

2

Background

In this chapter relevant background information and basic concepts are presented.

2.1 The Epiphany architecture

The *Epiphany* is a many-core processor from *Adapteva* [1]. It is available in a 16-core and a 64-core version. The Epiphany architecture can support up to 4096 cores. The cores are arranged in a grid, or 2D array, and connected with a *Network-on-Chip* (NoC). The NoC also connects the cores to external memory, a host processor, and possibly to other Epiphany processors.

Each core is a superscalar RISC processor with 32 KiB of local scratchpad memory split in four banks of 8 KiB [1]. Direct memory access is provided by a flat and unprotected memory architecture: all memory, local and shared SDRAM, and memory mapped peripherals are addressable in the same address space. Message passing is facilitated by a DMA engine.

The Epiphany's NoC is a 2D mesh that connects each core with its north, south, east and west neighbor. Three different meshes serve a specific type of traffic each: one mesh for on-chip write requests, one for off-chip write requests and one for all read requests. The three different meshes are depicted in Figure 2.1. The NoC is optimized for write requests, which are 16 times faster than read requests and non-blocking. The network is deadlock free since each mesh router operates in a round-robin fashion [1]. Traffic is routed first along columns and then along rows.

2.2 Inter-core communication

Inter-core communication can be achieved in at least two different ways on the Epiphany platform: through shared memory or message passing. Both methods use the same NoC

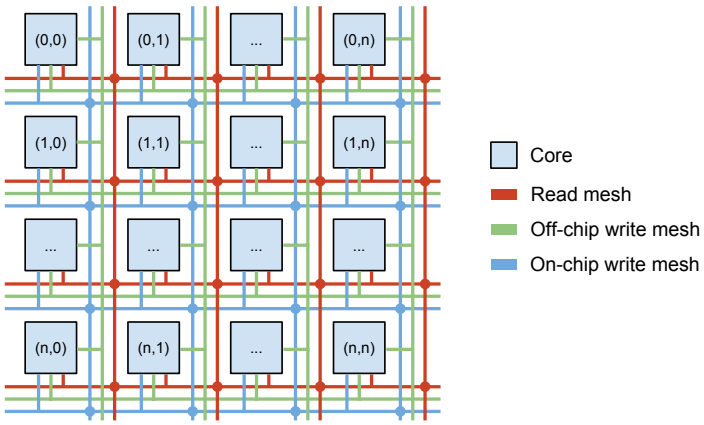


Figure 2.1: Overview of the three meshes in the Epiphany NoC.

and rely on accessing shared SDRAM or the local memory of another core. Message passing is an asynchronous method while shared memory is not.

2.2.1 Shared memory

A single memory space is used to address all memory on the chip, and by default all memory is readable and writable: every core can access the local memory of every other core [2]. The local memory of each core is addressable in two different ways: using a local memory address (starts at 0) or using a global memory address (prefixed with core ID.) The local address is typically used for local variables while the global address is typically used for shared memory (for accessing other cores.)

The same assembly instructions are used to access local memory and remote memory [1]. The hardware decoder decides if the memory address is destined for the local memory or if a network request is required. Write requests can be dispatched at the same rate as the core can produce them. Under ideal conditions writing will never stall the core.

2.2.2 Message passing

Asynchronous message passing is facilitated by *Direct Memory Access* (DMA) engines. Each core contains two DMA engines which are dedicated hardware for reading and writing remote memory without involving the processor pipeline. The processor can schedule reads and writes on the DMA and continue with other tasks, but a startup cost is payed when scheduling requests. Interrupts are used to signal the processor when DMA requests are completed. The DMA engine can transfer 64 bits per clock cycle [1].

2.3 The Parallella board

The *Parallella* board contains a Zynq dual-core ARM Corext A9 processor with FPGA logic, 1 GiB of RAM, an Epiphany co-processor with 16 or 64 cores, and common peripherals such as Ethernet and HDMI [3]. The FPGA is used for glue-logic between the ARM and Epiphany, a HDMI controller and for an AXI interface [2].

Henceforth the ARM processor will be referred to as the *host* processor and the Epiphany will be referred to as the *device*.

The host processor runs a Linux operating system while the device is used as a co-processor. The host processor is responsible for loading programs onto the Epiphany and for other control tasks. It is also responsible for linking programs and for initializing their run-time environment prior to launch. During initialization the host processor can write initial data to each core's memory and compute addresses of shared memory.

2.4 ESDK programming model

The *Epiphany Software Development Kit* (ESDK) is a collection of tools and libraries specifically crafted for programming the Epiphany processor [4]. The Epiphany is programmed using C and the ESDK provides API-functions for common tasks such as restarting the device, reading and writing device registers and memory, issuing DMA requests etc. The *Parallella Architecture Library* (PAL) will eventually replace the ESDK but as of March 2015 PAL is not ready for experimental use.

Epiphany programs are split into two parts: one part for the host processor and one part for the Epiphany. It is therefore necessary to write two or more separate programs: one that execute on the host processor, and one or more device programs that execute on the Epiphany [2]. The host program loads each device program onto the Epiphany, initializes its run-time environment, provides input data and performs other control tasks. The device programs are intended to execute computationally intensive code.

The ESDK provides programming primitives such as mutexes, timers and barriers [2]. It also defines other important concepts such as workgroups.

A workgroup is a group of cores that occupy a rectangular area on the Epiphany grid. Figure 2.2 depicts an Epiphany processor with a workgroup of four cores. Workgroups are created to control multiple cores simultaneously from the host processor and to group cores that perform related tasks. Workgroups are created by specifying the number of rows and columns, and the absolute location of the top-left core [4]. Cores within the same workgroup can query for the ID and grid location of neighboring cores [2].

The ESDK provides two sets of features: one set for the host and another set for the device. The host API offers functions for loading programs onto the device, for creating and managing workgroups, for reading and writing device registers and memory, for creating regions of shared memory in SRAM, and other control functions for managing the Epiphany. Most of these features are not available on the device. Instead, the device API offers functions for querying about the current and neighboring cores, for issuing DMA requests, for reading and writing SRAM, and other helper functions for writing

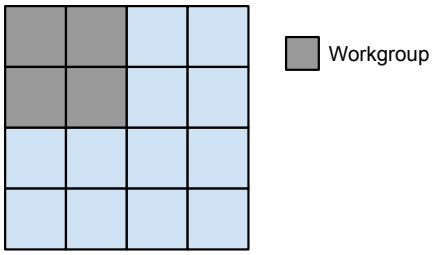


Figure 2.2: A 16-core Epiphany processor with a workgroup in the top-left corner.

parallel programs.

2.5 Program launch steps

The host processor will normally perform the following steps to launch an application on the Epiphany [2]:

1. Create a workgroup.
2. Reset all cores within the workgroup.
3. Load a device program onto every core within the workgroup.
4. Communicate with the device to set up the run-time environment. (Optional)
5. Start executing the application.
6. Communicate with the device.
7. Wait for the application to finish.

2.6 Allocation, assignment and remapping

Many-core applications are partitioned into smaller parts. These parts are usually refereed to as *tasks*. A task is a self-contained unit, a function, an algorithm, a state-machine, that operates on input data and produce output data. Tasks can have dependencies: they are not allowed to start until their input data is available or until a shared resource can be accessed.

Cores are allocated for each task. Some tasks are serial and will only receive a single core. Other tasks are parallel and multiple instances of the same task are executed on multiple cores simultaneously. System timing and throughput requirements determine how many cores are allocate for each task. Core allocation is assumed to be static for the purposes of this thesis.

Partitioning and allocation are determined by requirements and design goals. They are application specific and do not change. How an application is partitioned, will

affect everything about its design, while system timing and throughput requirements will determine how many cores are allocated for each task. Task assignment however, can be dynamic.

During assignment, a core is assigned to each task. The assignment process takes communication patterns, core features, and application requirements into account. When assigned, or mapped, the application can be executed on the target hardware: each core knows which task to execute. Unless hardware faults occur, or application requirements change, the same mapping is used indefinitely.

Remapping becomes necessary when hardware faults occur since faults render hardware resources unavailable and force remapping by invalidating the previous task assignments. Faulty cores are considered unavailable and cannot be used. Unless cores with special features become unavailable it should be possible to replace the faulty cores with spare cores to solve the problem. Since cores cannot be physically replaced, remapping instead reassigns the tasks. Tasks assigned to faulty cores are simply reassigned to fault-free cores. Remapping therefore targets core assignment and tolerates faults by reassigning the application.

Typical Epiphany applications are statically assigned by a designer or programmer. Their assignment does not change and does not need to change. But when fault tolerance is considered this is no longer true: static assignment cannot be used. An automated assignment process is required.

2.7 Writing relocatable software

Relocatable software is one of the main requirements of remapping. By relocatable we mean software that works on any core, it can be relocated from one core to another. The software should make no assumptions about its own grid location or that of other tasks. Aside from special core features the challenge when writing relocatable software is computing addresses for inter-core communication.

Inter-core communication on the Epiphany platform is conducted through memory accesses. Every core is accessible through a shared address space, and communications are simple memory reads and writes. The memory address of the target core is therefore required. There are essentially two different methods of acquiring the target address: it is either computed on the device or computed on the host processor. These two methods pose different requirements on the remapping system.

2.7.1 Device computed addresses

Knowledge about task assignment is necessary when computing target addresses. Tasks will typically communicate with specific tasks: for example, task A will communicate with task B, but not with task C. Task A must therefore know on which core task B is located. This information is transfered from the host to each core during startup.

The ESDK does not consider fault tolerance; therefore, many API functions and C-structs cannot be used with remapping. These structs and functions return information

without taking remapping into consideration. The functions will return locations of faulty cores.

Software that use these structs and functions are not remapping safe:

- `e_group_config` — struct with information about the current core and workgroup.
- `e_neighbor_id()` — returns ID of neighboring cores within the same workgroup.
- `e_get_coreid()` — returns ID of current core.

Similar functions and structs that are remapping aware are easily implemented. However, they require device memory for code and data. Since memory is scarce it could be preferable to let the host compute all memory addresses. On the host there is an abundance of memory and the required task assignment information is already available.

2.7.2 Host computed addresses

The host is better equipped than the device to compute addresses for inter-core communications. Firstly, task assignment is performed on the host and it therefore knows the location of each task. Secondly, the host exercises complete control of device memory prior to launch. In conclusion, the host has complete knowledge of the system and also exercises complete control.

During initialization the host writes an initial run-time environment to each core. This environment contains initial input data for each task, control flags and pointers to shared memory. All communication with other tasks is performed through these pointers, which typically point to *mailboxes*. Mailboxes are structs with shared data, such as integers, arrays and pointers. There is no need for the device to compute new addresses since the mailboxes contains all necessary information, which the user is responsible for providing.

Device memory is saved by computing addresses on the host. Only the information deemed necessary by the user is stored on each core.

2.8 Fault tolerance

A fault tolerant system continues to deliver correct service in the presence of faults [5]. Correct service usually means that no errors are allowed to occur on the system output: when observed from the outside the system appears to work correctly.

Avizienis et al. [6] defines the terms *failure*, *error*, and *fault*. A *failure* is an event that causes the system to deviate from correct service. The deviation is called an *error*. The cause of the error is called a *fault*.

Fault tolerance is typically achieved by adding redundancy [5]. The redundancy can be *passive* as in *Triple Modular Redundancy* (TMR) which masks faults without action from the system or operator. The redundancy can also be *active* (or *dynamic*). Active redundancy relies on reconfiguration to tolerate faults. Hybrid approaches are also possible with both passive and active redundancy.

Fault tolerance dictates that faults are contained: they are not allowed to propagate through the system. One failing component does not affect other components in other *fault containment zones*. In many-core processors each core is a fault containment zone. One faulty core does not affect neighboring cores; consequently, a faulty core is replaceable. Guaranteeing the integrity of fault containment zones is challenging. There are always some faults that propagate through the system: faults in the clock network is one example; faults in the network-on-chip is another.

Breaches in fault containment zones are tolerated if those breaches only affect a limited number of neighboring cores, that is, if the neighboring cores successfully trap the fault and stop it from spreading. When observed from the outside the neighboring cores will appear faulty, when in-fact, only one of the cores are.

2.9 Common sources of faults

Permanent hardware faults occur in the factory and during run-time. Manufactured faults are the results of fabrication defects and variations in the fabrication process. Process variations also give rise to wear-out effects [7]. Chips that are not 100 % functional are traditionally discarded [8], but this is about to change due to high production costs and increasing number of defects.

Transient faults are temporary malfunctions. They are short lived and disappear without causing permanent damage [9]. However, transient faults are 30 times more likely to occur than permanent faults during run-time [9].

2.9.1 Yield enhancement

ASIC manufacturing costs are largely driven by yield. Faulty chips are discarded and do not generate revenue. Maximizing yield is therefore an important aspect of maximizing revenue.

Significant increase in yield can be achieved by adding a few redundant cores to a many-core chip [10], and consequently, by allowing faulty cores on the chip. But redundancy is not without implications: adding redundant cores will increase chip area and decrease the number of chips per wafer. The amount of redundancy (the number of redundant cores) will therefore control yield.

Processors with eight or more cores will benefit from redundancy in the model presented by Andersson [10]. The largest number of healthy cores per wafer is achieved with 10 – 20 % redundancy. For a 64-core chip this represents 6 to 13 redundant cores, or approximately an additional row of cores in an 8 by 8 grid. This result assumes that R redundant cores are added to an N-core chip. If instead, R faulty cores are allowed in an N-core chip, the highest yield is achieved by allowing 6 to 11 faulty cores in a 64-core chip. This translates into a maximum of about 700 faulty cores for a 4096-core chip. The number of faulty cores will decrease as the fabrication process matures.

Instead of replacing entire cores others have suggested that redundancy is added at a finer granularity, such as by duplicating components [8], or by cannibalizing pipeline-

stages of other cores [11]. Others have suggested that virtualization is used to emulate faulty hardware resources at the expense of performance [12]. Redundancy at finer granularity may require redundant hardware and additional logic to deal with hardware reconfiguration. Adding redundant hardware will increase the complexity and cost of each core, reducing the advantage of simple cores [12].

The permanent hardware faults that are introduced during manufacturing can be dealt with using passive redundancy. By burning fuses the hardware is permanently reconfigured to mask the faults, rendering the faulty cores invisible to the user. Hardware support is required for this kind of masking and permanent reconfiguration. IBM tried this approach in their Cell processor for the Sony PlayStation 3. To increase yield one of the Cell's eight cores was disabled [13].

Instead of masking faults by reconfiguring hardware they can be dealt with by reconfiguring the user software. Remapping takes a software-only approach to reconfiguration.

2.9.2 Wear-out

Wear-out defects are permanent hardware defects that occur during run-time in integrated circuits due to aging mechanisms. They are caused by fabrication process variations [7], negative bias temperature instability (NBTI), electron migration, hot carrier induced damage, temperature cycling etc. [14]. These defects cause intermittent faults and permanent faults.

Stan and Re [14] propose that reliability sensors are added to the design. These sensors track NBTI and electron migration aging, and indicate when catastrophic failure is imminent. The need for large design margins is avoided, and hardware diagnosis fault tools can focus their analysis on a smaller chip area.

2.9.3 High energy particles

Transient faults occur when high energy particles strike transistors, wires, and memory elements. These particles cause a momentary voltage spike that can flip register bits and mimic digital pulses. They do not cause permanent defects, but can alter register values and cause incorrect program execution [15]. Transient faults are also caused by voltage fluctuations and electromagnetic interference [9].

3

Remapping

In this chapter dynamic core assignment, or *remapping*, is investigated as a fault tolerance method.

3.1 Problem description

The underlying goal of this thesis is to investigate remapping as a means to achieve fault tolerance in many-core processors. Fault tolerance in this context includes tolerating fabrication defects introduced to increase yield and permanent hardware faults that appear in the field. Which faults that can be tolerated and how remapping affects the system is discussed below.

Remapping poses requirements on the application software, the surrounding system and the layout. We investigate which these requirements are and how they affect the application and its performance.

Different core assignment schemes will affect the performance in different ways. In the simples scheme, a few cores are designated as reserve cores. When a core becomes faulty it is replaced by one of the spare cores. The distance between the faulty and spare core will affect communication latencies. The application's communication pattern will determine how much the performance deteriorates. We will therefore investigate different assignment schemes.

3.2 System requirements

A system with remapping support is assumed to run relocatable software and have the ability to detect and diagnose hardware faults.

3.2.1 Error detection and hardware diagnosis

The ability to detect and diagnose hardware faults is required since knowledge of faulty cores is required when remapping the user application. Without this knowledge the application cannot be remapped to avoid faulty cores. Consequently, remapping relies heavily on the accuracy of this information.

Error detection can be implemented using software tests and diagnosis — a low-cost option — or built-in self tests (BIST) [12]. The choice of error detection mechanism will affect run-time performance. However, the performance of the diagnosis mechanism is not important [12]. Diagnosis is only performed when new hardware faults are detected; this is a rare event that will cause remapping, and remapping is a slow process that will restart the application.

It is not strictly necessary to restart the application when errors occur. The affected core could be disabled and its task reassigned to another core. This requires that pointers to the failing core's memory are recomputed and updated. Data will be lost while reconfiguring affected tasks, but the application can continue without rebooting. This technique is beyond the scope of this thesis. We will assume that the application is restarted.

3.2.2 Relocatable software

Assumptions about available cores must not exist in the user application. Remapping will assign different tasks to different cores on different processors. Writing the application to be core agnostic is therefore important.

Writing core agnostic software can be easy or challenging depending on the architecture and API of the platform. On an Epiphany processor it is possible, and easy, to reference the memory of other cores using hard-coded addresses; this is bad practice and not compatible with remapping. Instead of referencing cores by their memory addresses they can be references indirectly through ID-numbers returned by API functions. API functions can for example query for neighboring cores or allow the programmer to iterate over every core assigned to a specific task.

Another solution is to store a look-up table, or a C-struct, in each core with pointers to shared data. The pointers are set by the host processor prior to launching the device program.

3.3 System overview

The system consists of three parts: firstly, the error detection and hardware diagnosis tool; secondly, the remapper and associated API; and thirdly, the user application. These three parts work in unison to produce a fault tolerant system.

Upon system startup the error detection and hardware diagnosis tool is executed. This tool finds hardware faults, runs diagnosis and outputs a fault report. This process is presumably slow. Afterwards the user application is launched. This application invokes the remapper which reads the fault report and remaps the application accordingly. This

process is also slow if new faults have occurred since last launch. When mapped, the user application is allowed to execute on the device. If a fault occurs during execution the user application is halted and the system is rebooted.

3.4 Tolerated faults

From Section 2.9.1 we know that fabrication defects are common. Tolerating permanent hardware defects is therefore important; but not all defects cause faults that can be tolerated by remapping, and remapping may tolerate faults that are not caused by permanent hardware defects. Which faults can be tolerated by remapping? What faults require another strategy?

Remapping tolerates faults by not using the faulty hardware. Remapping can only tolerate faults at core granularity since it works by reassigning tasks at the core level: faulty cores are not used after remapping. This means that a single defect in a single memory bank will render the entire core useless. Hardware is sacrificed in favor of simplicity and in order to achieve a software-only solution. With a more complex remapping scheme — that knows about the application's memory usage — it might be possible to use only memory banks that are known to work while avoiding faulty banks. Other methods may use redundancy within each core, such as multiple ALUs, to increase reliability [12] but these methods are beyond the scope of this thesis.

3.4.1 Permanent hardware faults

Permanent hardware faults can occur in two different scenarios: either during fabrication or in the field. Permanent faults that occur during fabrication are easier to handle than those that occur in the field. We can assume that fabrication related defects are contained, well behaved and that they affect a limited number of cores. Preferably they are also well documented and well understood. It lies in the manufacturer's interest to assure customers that fabrication related defects can be dealt with. However, defects that occur in the field do not follow the same rules. They can affect the entire processor, disrupt clock signals, waste power, or make the entire processor unreliable.

The fabrication process must contain a rigorous test, which will detect and document chip defects. Chips that are deemed *not-too-broken* are sold, and not-too-broken could mean different things in different price ranges. For military applications it might mean fully functional, and for consumer products it might mean up to 20 % of faulty cores.

Regardless of price range there are only some faults that can be tolerated with a software-only remapping solution. The tolerated faults are limited to well behaved and well confined faults that only affect a single core, or appear to only affect a single core. This means that defects in the clock network, in the network-on-chip, or other global resources, cannot be tolerated. However, faults in core memory, pipeline, register file, or other local resources, are most likely tolerated. Stuck-at faults, shorted wires, and broken wires, are most likely tolerated, if contained in a single core.

Wear-out faults are different from fabrication faults: they occur in the field, during

run-time. While fabrication related faults are limited to fault containment zones, wear-out faults are not. They can occur anywhere in the hardware and cause any type of error. Tolerating wear-out faults is therefore more challenging, nevertheless, remapping is a viable option if fault containment zones are intact. When used in conjunction with reliability sensors, wear-out faults can be avoided before they occur, increasing the chance of keeping fault containment zones intact.

Meixner and Sorin [12] report that their hardware virtualization technique *Detouring* covers 42.5 % of all hard stuck-at faults that can occur in a simple RISC OR1200 core. The majority of covered faults are located in the register or multiplier. Detouring is limited in what it can accomplish: it tries to mask defects by running software on the faulty core. Remapping tries not to utilize faulty cores and may therefore have a higher coverage for hard stuck-at faults within each core.

3.4.2 Transient faults

Remapping is not well suited to tolerate transient faults. Transient faults occur often and should therefore be dealt with when they occur to recover the state of the application. Remapping is too slow and invasive to handle transient faults. However, remapping can disable and avoid cores that experience abnormal amounts of intermittent faults.

4

Workgroup mapping

This chapter will introduce a remapping system based on the models and concepts found in the *Epiphany Software Development Kit* (ESDK). It will do so in the context of Adapteva's two dimensional FFT-example application.

There are multiple reasons for choosing Adapteva's FFT example: the code is open source and freely available; FFT is a common application well suited as a small accelerator; and it produces an easily verifiable result, an image. But before we dwell into the details of Adapteva's two-dimensional FFT example we will first discuss an important concept found in the ESDK: the workgroup.

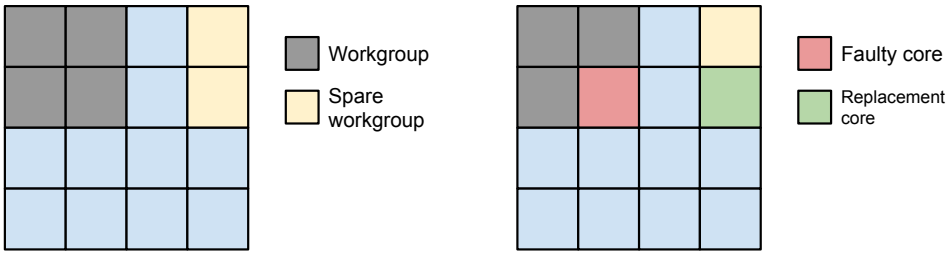
4.1 Workgroups

Our first attempt at designing a remapping system will follow the methods and concepts found in the ESDK. Following established methods and concepts will make it easier to design the system and for others to adopt it. One of those concepts is the *workgroup*.

Workgroups are used to operate on multiple cores simultaneously and to group geographically related cores. For example, a single API function call can load the same program onto all cores within the same workgroup. For remapping purposes the workgroup will be used to specify groups of related cores and groups of spare cores. When remapping, faulty cores are replaced by spare cores from another workgroup called a *sparegroup*.

In Figure 4.1a the workgroup in the top-left corner will use spare cores from the sparegroup in the top-right corner if necessary. The size and position of groups are specified by the user. In Figure 4.1b a faulty core is replaced by a spare core.

The workgroup is a central concept in the ESDK and many API functions are specialized around them. One of those functions is `e_neighbor_id()`. It returns the grid location of neighboring cores — next or previous — within the same workgroup. This function does not work in remapping applications, neither does other ESDK functions



(a) Sparegroup introduced in the top-right corner.

(b) A faulty core is replaced by a spare.

Figure 4.1: A 16-core Epiphany processor with a sparegroup and a faulty core.

oblivious to remapping. However, `e_neighbor_id()` and friends, is commonly used to compute memory addresses to shared data. To emulate their behavior a virtual coordinate system is used. This coordinate system masks faulty by mapping virtual grid locations to physical grid locations. More specifically, grid locations of faulty cores map to spare cores. This allows API-functions to behave as if not faults exist, simulating the behavior of equivalent ESDK functions.

4.2 Remapping process

The remapping process is split into three stages:

1. Assign a spare core to each faulty core.
2. Compute global memory addresses to shared data.
3. Load program code onto each core and run the application.

During stage 1 a task is assigned to each core while faulty cores are avoided and replaced by spare cores. A virtual map is created in this stage, this map translates virtual grid coordinates into hardware grid coordinates. The virtual map is important for later stages as it describes where cores are located in the hardware; this information is used to compute global memory addresses in stage 2. During stage 2 the initial run-time environment is written to each core. This includes memory addresses to shared data and other values necessary when each task is first executed. No code is running on the device during stage 2, the host processor can manipulate device memory without interference. During the third and final stage program code is loaded on each core. The program code should correspond to the run-time environment prepared in stage 2. After stage 3 the application is running on the device.

When remapping, the host processor is responsible for assigning a task to each core. During the assignment process it avoids faulty cores and adheres to design constraints. An external fault detection and diagnosis system provides the necessary information about faulty cores, and the assignment algorithm avoids these cores to provide fault

tolerance. Some applications may for instance require that a specific task is only assigned to cores along the left-most column. Such requirements, and other design constraints, should be considered by the assignment algorithm. The host is also responsible for computing memory addresses of shared data and for relating those addresses to concerned tasks. These responsibilities can be summarized in a list:

- Assign a task to each core and avoid faulty cores by utilizing information from a hardware fault detection and diagnosis system.
- Adhere to design constraints.
- Store assignment results for use when computing global memory addresses.

4.2.1 Adhere to design constraints

Design constraints are described through workgroups. Workgroups limit where tasks are located in the grid and limit which spare cores to choose from. Spare cores are only chosen from a single sparegroup when remapping a workgroup; the designer can therefore associate different workgroups with different sparegroups. In Figure 4.2 workgroup A can be configured to pick spare cores from either sparegroup A or B but not from both. The designer can specify where spare cores are located, how many that exist, and which spare cores to choose from when mapping a workgroup.

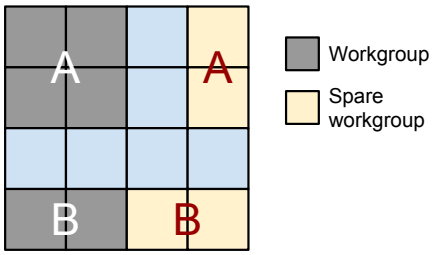


Figure 4.2: Workgroups and sparegroups can be combined arbitrarily. Workgroup A could use spare cores from either sparegroup A or B.

4.2.2 Avoid faulty cores

A fault map, or fault matrix, is used to avoid and disable faulty cores. The fault matrix contains an element for each core in the system. Each element describes the health of a core; it could for example indicate that memory bank zero is faulty. For the initial implementation the element simply indicates if the core is faulty or not. All faults are considered fatal and only 100 % non-faulty cores are used. The fault matrix is filled with the results of the fault detection and diagnosis system prior to remapping the application; it could be read from a file, standard input, or set directly in code.

4.2.3 Store assignment results

Faulty cores are replaced by spare cores on a workgroup granularity. During assignment the location of each assigned core is stored within the workgroup. All cores within a workgroup are therefore known, even if some cores are replaced. This information is used to compute global addresses.

4.3 Implementation

This section will discuss how the sparegroup concept is implemented. The implementation is split in a host part and a device part. The host part executes the remapping algorithm while the device part is used to query about remapping information for the current core.

4.3.1 Fault matrix

The fault matrix holds the health status of each core. The fault matrix is represented as an array of unsigned integers. Each element is a bit field where each bit represents a particular fault. If all bits are cleared the core is considered non-faulty and will be assigned a task. A core is considered faulty if any bit is set. Faulty cores are unused and replaced during assignment.

Bit 0 represents an unknown fault while bit 1 could for instance represent a fault in memory bank 1, and bit 6 a fault in the network router etc. Table 4.1 shows an example interpretation of the bits in each matrix element. In this implementation we do not consider different faults even if they are representable in the fault matrix. The assignment algorithm is only interested in distinguishing between faulty and healthy cores. A more clever assignment algorithm could use the detailed fault information to utilize partially faulty cores.

Table 4.1: Example interpretation of the bits in a fault matrix element. If a bit is set the core contains the described fault.

Bit	0	1	2	...	5	6	...	31
Fault	Unknown	Memory Bank 1	Memory Bank 2		DMA	Router		

The fault matrix representation was designed for simplicity. It should be possible for programs to generate applicable fault matrices with ease. For a 64-core Epiphany chip the fault matrix is represented using 64 unsigned integers. These integers can be read from a file, from standard input or set using appropriate API-function. The integers are stored as a continuous array where element 0 represents core $(0, 0)$, element 1 core $(1, 0)$, element 8 core $(0, 1)$ and so on as seen in Table 4.2.

Example: A system consists of two programs. Program A is a hardware fault diagnosis tool and program B implements a remapping application. When a fault is detected

Table 4.2: Fault matrix indices for different core locations.

Array index	0	1	2	...	7	8	9	...	63
Grid location	(0,0)	(0,1)	(0,2)	...	(0,7)	(1,0)	(1,1)	...	(7,7)

the system is restarted. During system startup program A is executed to diagnose any hardware faults. Program A outputs a file called *hwfaults*. This file contains 64 unsigned integers, non-faulty cores are marked with zeros. When program A exists program B is started and instructed to read the *hwfaults* file and load all integers into its fault matrix. Remapping now avoids faulty cores detected by program A.

4.3.2 Workgroup

The workgroup represents a rectangular area on the Epiphany chip. It also represents a mapping from faulty to spare cores. For each core within the workgroup a virtual and a device location is stored. The virtual location represents the core's location in a non-faulty chip, and the device location represents the core's true location in the chip. For most cores the virtual and device location is equivalent, but for faulty cores the device location will point to a spare core.

For a remapping system to be functional and useful the device application should be oblivious of its existence. This means that regardless of mapping the device application should work without modification. An abstraction layer between the device and the application mapping can solve this issue. By referencing cores using virtual coordinates instead of device coordinates the mapping will always appear identical. The ESDK relies heavily on similar abstractions: for example, most API-functions use relative coordinates, well written code will work regardless of group location since $(0, 0)$ always refers to the first core within the workgroup. For remapping purposes the virtual coordinates will span a continuous region of non-faulty core locations. Contrary to an ESDK workgroup all locations within a remapping workgroup will reference non-faulty cores.

4.3.3 Sparegroup

Similar to workgroups, the sparegroups also represent a rectangular area on the Epiphany chip. For each spare core within the sparegroup a device location and a flag is stored. The flag is used to indicate if the spare core have replaced a faulty core or not. The assignment algorithm will search for unused spare cores by checking their flag.

4.3.4 Assignment algorithm

The assignment algorithm is deterministic: given the same input it will produces the same output, every time. Determinism allows a remapping application to execute using the same mapping until a hardware fault occurs. The fault will change the input of the algorithm (the fault matrix) and produce a new output (a new mapping.)

The assignment algorithm is straightforward: for each workgroup and sparegroup pair, iterate over every core in the workgroup, and replace every faulty core with the first free spare core in the sparegroup, and mark the spare core as non-free.

4.3.5 Translation table

The translation of the grid locations of a faulty cores into the grid locations of a spare cores is implemented as a table lookup. This translation is important when emulating functions such as `e_neighbor_id()` and other ESDK functions described in Section 2.7.1.

The translation table contains the grid location of every faulty core and the corresponding location of spare cores. Contrary to previously described concepts, the translation table is located on the device. Assuming that a maximum of 10 % of all cores are faulty the table will contain a maximum of seven entries for a 64-core chip. A brute force lookup that checks every entry will suffice. Each entry contains four bytes: two locations with two bytes each. The table will require 28 bytes of memory on each core for a 64-core chip, and 2.8 KiB for a 4096-core chip. Table 4.3 contains an example translation table. Remapping relies on the host processor to perform the task assignment and to write a translation table to each core during initialization.

Table 4.3: Translation table from faulty to spare core.

Grid location			
Faulty		Spare	
X	Y	X	Y
0	6	2	0
1	4	2	1
⋮			
1	6	2	4

The number of faulty cores that can be tolerated is limited by the table size. This can present a problem for applications with many spare cores. These application are limited by the table size, not the number of spare cores. An application that only uses a single core can in theory keep working until all cores have failed, except it will run out of table entires before that happens. However, vital shared hardware will most likely fail before all cores do. A configurable table size is preferable.

A unique translation table exists per workgroup. This means that each workgroup only knows about cores within the workgroup. Cores in other workgroups are inaccessible. Inter-workgroup communications are therefore impossible if global addresses are computed on the device.

4.3.6 Launch preparation

An application written using the ESDK requires seven steps to start running. It also requires that two separate programs are written: one for the host processor and one for the device. A remapping application faces similar requirements but additional steps are taken to launch the application:

1. Run hardware fault and diagnosis tool:
   (a) Identify and diagnose hardware faults.
   (b) Write results to disk.
2. Run remapping application:
   (a) Create an ESDK workgroup for all cores in the device.
   (b) Reset all cores in the device.
   (c) Create a fault matrix and populate it with faults from disk.
   (d) Create a workgroup and a sparegroup.
   (e) Assign spare cores to the workgroup.
   (f) Communicate with the device to setup the run-time environment.
   (g) Load the a device program onto every core within the workgroup.
   (h) Start executing the application.
   (i) Communicate with the device.
   (j) Wait for the application to finish.

4.4 Results

Adapteva's 2D-FFT application was remapped, and the execution time measured for different locations of a spare core. The core at $(0, 0)$ was replaced with a spare core at $(4 + n, 4 + n)$ where n is an integer. Figure 4.3 shows that the execution time decreases as the distance between the faulty and spare core increases. This is not what we expect.

A test application was executed under different conditions and configurations, to test if core-to-core distances and the workloads of surrounding cores are the most important contributers to execution time, when considering communication related contributions. The test application forms a chain where each core sends a small packet to the next core. The chain is circular and execution time is limited only by communication performance. To test the impact of distance, a single core is moved from its original location $(3, 3)$ diagonally downwards to $(7, 7)$; the inverse is also tested. For each configuration, different workloads are loaded on the surrounding cores. These workloads do not participate in the chain, but produce interference that degrade communication performance. The workload labeled *idle* issues the IDLE assembly instruction. *Internal* and *loop* runs an endless while-loop, but *internal* also reads from an internal register. *North*, *east*,

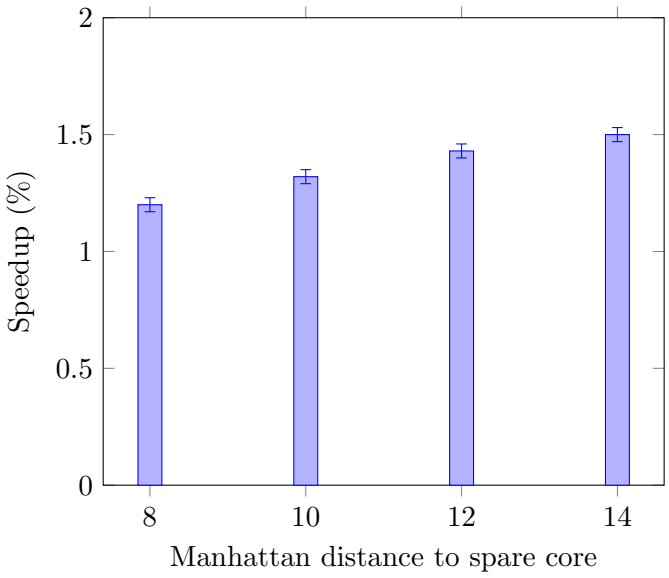


Figure 4.3: Speedup of introducing a spare core for $(0, 0)$ at different distances (diagonal.)

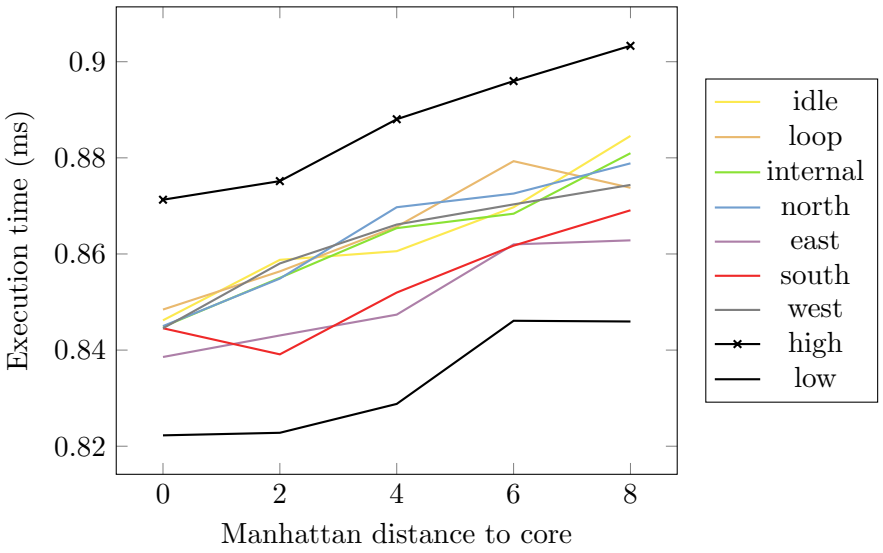


Figure 4.4: Execution time for different interferences and core distances (down-right.)

south, and *west* sends data packets in the indicated mesh direction. The results of this experiment are shown in Figure 4.4 and Figure 4.5 where *high* and *low* indicate a standard deviation. Execution time is affected by distance in both figures. An increase of up to 5 % is observed. Both figures display similar execution time regardless of workload on surrounding cores. Only north and east-bound traffic seem to affect performance, but the difference is well within the margin of error.

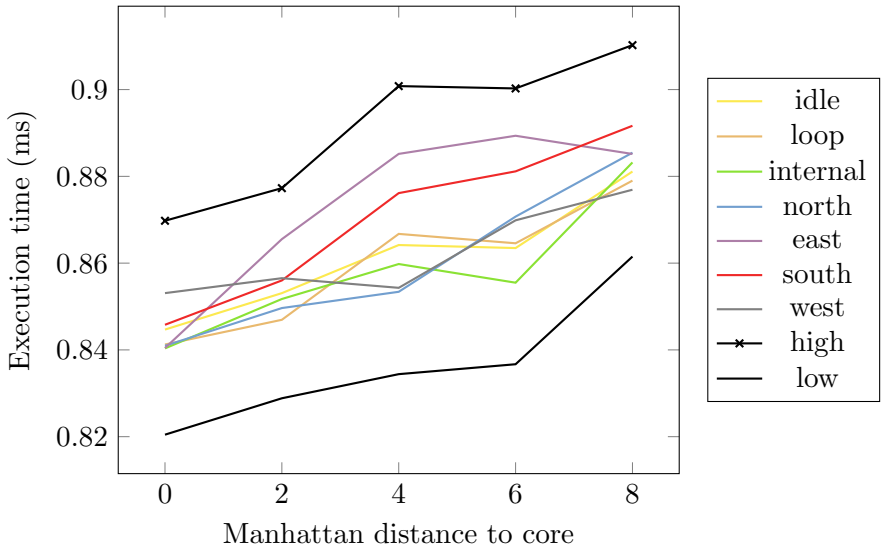


Figure 4.5: Execution time for different interferences and core distances (up-left.)

4.5 Discussion

The implemented assignment algorithm picks the first spare core it finds, regardless of distance between the faulty and spare core. As seen in Figure 4.4 this strategy hurts performance unless the sparegroup and workgroup are placed close together. The assignment algorithm could be improved by sorting the spare cores by distance before picking a replacement.

The test application used in Figure 4.4 and Figure 4.5 is communication bound and therefore greatly susceptible to degraded communication performance. In reality user applications are not communication bound, they are instead computation bound. Communication distances and surrounding workloads will have less impact on their performance than the test application presented here. Nevertheless, improving performance where possible (and easy) is always valuable, especially in high-performance many-core systems.

The results presented in Figure 4.3 are counter intuitive as execution time should increase when network distances increase. Longer distances result in longer latencies and therefore longer execution time. However, distance is not the only contributing factor. Surrounding cores will send data through the same network routers. As indicated by Figure 4.4 these surrounding cores will affect performance. The network routers operate in a round-robin fashion, which means that cores sending data through the same router will have to wait for their turn. Due to its implementation the arbiter is more likely to choose some directions more often than others. When a task is moved, this can greatly affect how routers prioritize the data, and consequently give rise to counter intuitive behavior.

Workgroup mapping essentially solves the problem of remapping. It can reconfigure

applications to avoid faulty cores, and keep them operational in the presence of hardware faults. It is however, not without limitations. The maps produced by the assignment algorithm will hurt performance. It does not try to minimize distances or interference; it simply replaces faulty cores with spare cores. Workgroup mapping is therefore not well suited for hardware with fabrication related defects where the application runs on faulty hardware for a long time. It is however, well suited for situations where the application runs on faulty hardware for a limited amount of time, such as while waiting for a service technician to replace it. Remapping with workgroups is near instantaneous: it only needs to swap two virtual coordinates. It can therefore quickly remap and restart the application while waiting for a service technician.

Remapping Adapteva's 2D-FFT example proved that adopting an existing application is relatively easy since the remapping API is similar to the ESDK API. This could help others adopt remapping.

The virtual coordinates allows the programmer to assume a static configuration, and to assume that tasks are located on specific coordinates. This property is otherwise only found in healthy hardware, and it can make it easier to program the system.

Computing global memory addresses is one of the most important steps when launching an Epiphany application. To compute global addresses it is necessary to manually (and mentally) keep track of all cores that execute a specific task. It is also necessary to iterate over all the related cores, column by column, row by row. This work is error prone and better suited for a computer. Finding an automated approach would greatly simplify programming and maintenance. However, workgroups and sparegroups are not suitable for such automation as they provide no useful data structures. Furthermore, workgroups do not store which task , nor how tasks are related, leaving the automated tool with little to work with. A more suitable approach and data structure is necessary.

Allocating many different tasks to many different cores is another important aspect that works unsatisfactory with workgroups. The API is best suitable for running a single task on all cores within the workgroup. This assumption is not well aligned with reality. An improved solution is introduced in Chapter 5.

5

Matrix mapping

The programming model introduced by workgroups was found to be unsatisfactory. This chapter will introduce a new programming model that aims to solve those problems. The new model will produce better results (in terms of performance) and code that is easier to write and maintain.

5.1 Moving away from workgroups

To explore the issue with workgroups, and to find a solution to those issues we will consider an example application. This example application is a pipeline consisting of four different stages of varying parallelization. The pipeline is visualized in Figure 5.1. Eight cores are necessary to implement the pipeline. Each stage in the pipeline performs computations on the data provided by the previous stage before forwarding that data to the next stage. The first stage receives data from the host, and the last stage collects the result, which is later read by the host. Data is sent by sharing memory: the host accesses the local memory of task 1, and task 4, while task 1 accesses the local memory of task 2 etc.

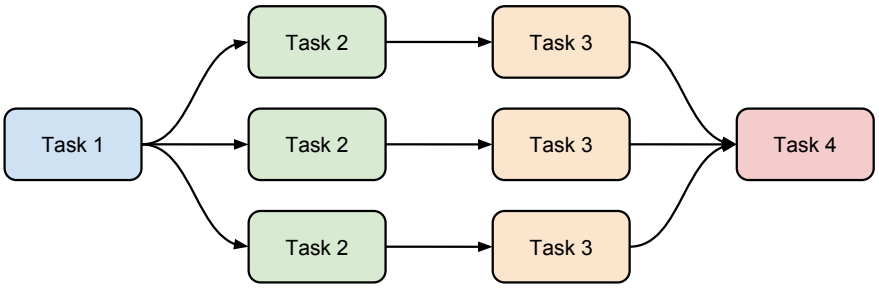


Figure 5.1: Example application pipeline. Arrows denote flow of data.

Data flows from stage to stage in the pipeline. This flow requires that each stage knows about the next, and by extension that each task contains a pointer to the local memory of the next task. In turn, this requires that someone knows the location of all tasks on the Epiphany grid. This was previously solved by burdening the programmer with manually keeping track of task locations in a virtual coordinate system. The programmer was also burdened to manually write glue-code for computing global memory addresses. This was especially daunting for applications with many tasks and many interconnections between tasks — in this context many is more than one. Matrix mapping tries to solve these issues by automating the process of task assignment and by simplifying the glue-code.

When using workgroups it is cumbersome to achieve the core allocation depicted in Figure 5.2. Workgroups describe a rectangular area in which multiple instances of the same task are allocated. The pipeline cannot be easily described using rectangles: at least three workgroups are necessary to describe the four different stages and an additional five sparegroups are necessary to cover all cores in the chip. The programmer will therefore have to manage a total of eight groups. This is unacceptable.

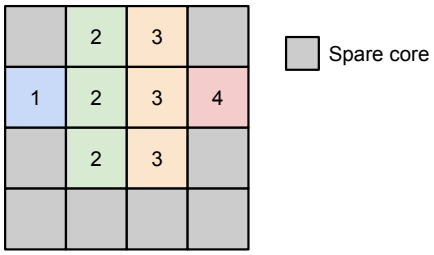


Figure 5.2: Example pipeline core allocation in 16-core Epiphany. Numbers denote tasks.

While studying Figure 5.2 we can realize that there is an obvious way to express the core allocation it depicts: by using a matrix identical to the figure. Instead of expressing the Epiphany grid as workgroups, we express the grid directly as a matrix. This matrix contains an element for each core in the device, thus, each matrix location maps to a core location. Non-empty elements are allocated for tasks, while empty elements are used as spare cores. An example *allocation matrix* is shown in Figure 5.3a.

To load a program onto the device using the allocation matrix, the loader will simply iterate over all elements in the matrix, and load a device program onto the corresponding core. However, since faults can occur an additional step is necessary: replacing faulty cores with spare cores. An assignment algorithm will pre-process the allocation matrix and produce a similar matrix, an *assignment matrix*, where faulty cores are no longer assigned. The assignment matrix is then fed to the loader, which loads the device programs.

The assignment algorithm implements remapping. It takes a programmer specified allocation matrix and transforms it into a remapped version, where faulty cores are no longer assigned. The remapped matrix is called an *assignment matrix* — it assigns a task to each healthy core. During assignment, tasks are moved if they are located on

a faulty core. If for example core $(1, 3)$ — which allocates a single core to task 4 in Figure 5.3a — is faulty it will be reassigned; Figure 5.3b shows a possible solution where $(0, 0)$ is used instead.

	0	1	2	3
0		2	3	
1	1	2	3	4
2		2	3	
3				

(a) Allocation matrix

	0	1	2	3
0	4	2	3	
1	1	2	3	-
2		2	3	
3				

(b) Assignment matrix. $(1, 3)$ is faulty.

Figure 5.3: Core allocation and assignment matrix based on Figure 5.2. Spare cores are denoted by empty cells and faulty cores by dashes.

There are usually many spare cores to choose from when reassigning a task. Some spare cores are far away, and others are closer to the faulty core. Choosing a spare core close to the faulty core should result in minimum performance change. A naive assignment algorithm will not consider this, it will simply choose the first spare core it encounters. A smarter assignment algorithm will try to minimize the distance between faulty cores and their spare cores. However, producing an optimal solution is difficult.

5.1.1 Implications

Using matrices to express task allocations and assignments have implications for the programs we write. Since the ESDK uses workgroups and relies in the information they provide, most ESDK function will not work as intended. This was also true for remapping using sparegroups, but in that case it was possible to simulate the behavior of affected ESDK functions. It will not be possible this time; at least not with reasonable effort. Also, matrix remapping tries to move away from workgroups and the issues they cause. Simulating ESDK behavior is therefore not a goal.

Information about neighboring cores is lost with matrix mapping. It is still possible to query the ESDK for neighboring cores, but the returned information will not consider remapping and it will not be possible to wrap at workgroup borders. It will neither be possible to translate the core ID and core location into a virtual, continuous, coordinate system as was possible for sparegroup mapping. Therefore, the device program is effectively oblivious to its surroundings. A core cannot know what tasks are running on neighboring cores, and it cannot know where to find a core running a specific task. This will free device memory, and make room for more input and output data.

The implication of oblivious device programs is simple: all global addresses must be computed on the host. Only the host knows where tasks are located and how tasks relate to each other. In the pipeline example, only the host known enough to compute the global addresses required by task 1 when it communicates with task 2 for instance.

5.1.2 Enforcing design constraints

The assignment algorithm discussed so far does not consider location based design constraints; it only considers the number of cores assigned to each task and their fault-free location. It does not consider locations with special features or special meaning to the application. For example, cores on the first row could support broadcasting while other cores do not. In the algorithm discussed so far it would be impossible to limit task 4 to the right-most column. When a fault occurs in $(1, 3)$ task 4 can be moved to any spare core.

A *constraints matrix* is introduced to limit the choices of spare cores. The constraints matrix is similar to the allocation and assignment matrix in that it contains a single element for each core, but it is also different since multiple constraints matrices exist: one for each task. The constraint matrix describes which cores to choose from when replacing a faulty core on a per task basis. Figure 5.4b shows an example constraints matrix that constraints task 4 to the right-most column. Figure 5.4c is the resulting assignment matrix when a fault is introduced at $(1, 3)$.

	0	1	2	3
0		2	3	
1	1	2	3	4
2		2	3	
3				

(a) Allocation matrix

	0	1	2	3
0				4
1				4
2				4
3				4

(b) Constraints matrix

	0	1	2	3
0		2	3	4
1	1	2	3	-
2		2	3	
3				

(c) Assignment matrix

Figure 5.4: A constraints matrix forces task 4 to be assigned to the right-most column.

The constraint matrix is only considered when a faulty core is replaced. It cannot be used to move tasks assigned to healthy cores. If the allocation and constraints matrices are in disagreement, the allocation matrix location wins as long as that location is non-faulty.

The chance of successfully mapping the application decreases when constraints are introduced. Constraints limit the assignment algorithm's choice of spare cores, and therefore increase the risk of failure. Mapping can become impossible when faults occur if the constraints are too harsh. They also increase the execution time of the assignment algorithm as described in Section 5.3. Constraints should be considered a last resort and only used when necessary.

5.1.3 Putting it all together

Only one matrix is missing: the *fault matrix*. As before the fault matrix contains all faults that exist in the system. This matrix is populated by an external hardware fault detection and diagnosis system.

	0	1	2	3
0		2	3	
1	1	2	3	4
2		2	3	
3				

(a) Allocation

	0	1	2	3
0				4
1				4
2				4
3				4

(b) Constraints

	0	1	2	3
0				
1				1
2				
3				

(c) Faults

	0	1	2	3
0		2	3	4
1	1	2	3	
2		2	3	
3				

(d) Assignment

Figure 5.5: All matrices used in matrix remapping.

All matrices used by the matrix remapping system are found in Figure 5.5. The allocation and constraints matrix is provided by the designer, while an external tool produces the fault matrix. The assignment matrix is produced by the assignment algorithm which takes the other three matrices as input.

5.2 Assignment algorithm

The assignment algorithm is responsible for replacing faulty cores with spare cores. It takes the allocation, constraints and fault matrix as input and produces the assignment matrix. It was previously stated that a naive assignment algorithm will replace a faulty core with the first spare core it finds, but is there a better way? Can the assignment algorithm aid in computing global addresses? Or better yet, can the assignment algorithm utilize task connectivity to produce better results and reduce the programmer's burden?

5.2.1 Minimizing distance

Figure 5.6a shows the pipeline's core allocation: the figure represents an allocation matrix where empty cells denote spare cores, and numbered cells denote cores allocated for the corresponding task. More information is required to implement a clever assignment algorithm; specifically, the connectivity between tasks is missing. Figure 5.6b shows the connections between pipeline tasks, where nodes represent tasks and edges represent data flow.

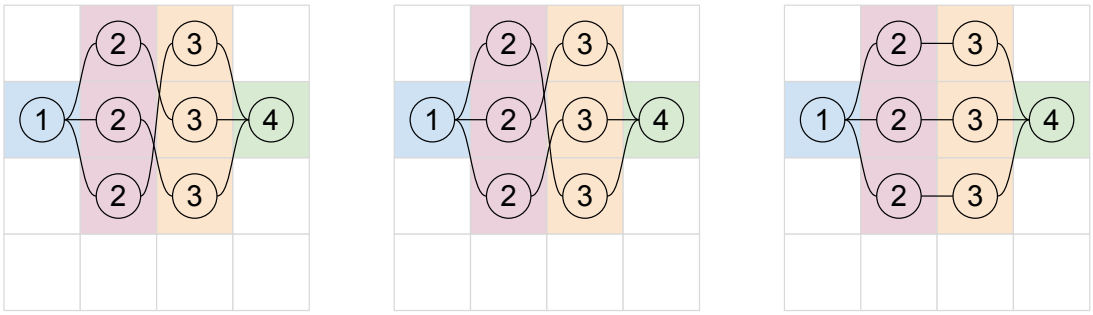
As shown by Figure 5.7 the assignment algorithm can produce different results by simply interchanging different task instances. Some of these results are better than others. In Figure 5.7a the distances between instances of task 2 and task 3 are unnecessarily large when compared to Figure 5.7c.

Figure 5.8 shows three different assignment solutions. The first is a naive solution, where the first encountered spare core is picked. The second tries to minimize the distance between the faulty and spare core. The third tries to minimize the distances between communicating cores.

From Figure 5.7 and Figure 5.8 it becomes evident that connectivity should be taken

(a) Task allocation **(b)** Connectivity graph

Figure 5.6: Connectivity graph and allocation matrix for the pipeline application.



(a) Bad **(b)** Bad **(c)** Good

Figure 5.7: Application connectivity depicted in allocation results.

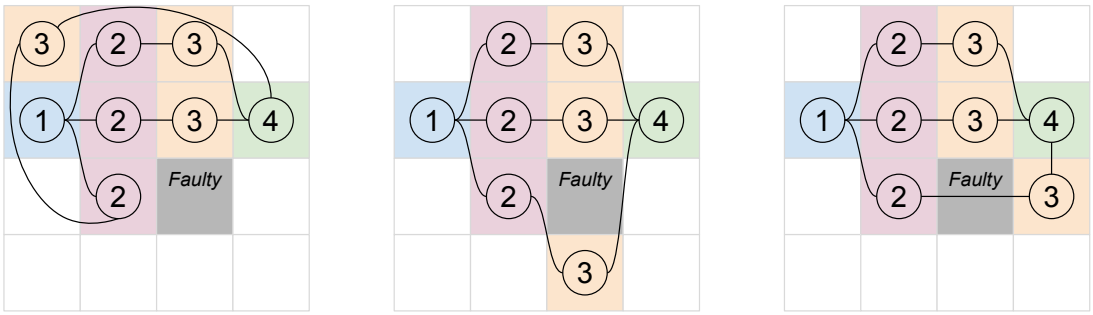
into account during assignment. If the distances between connected cores is minimized the application will experience less network latency and less network interference from unrelated tasks. An optimized assignment should result in better performance.

Optimizing the assignment can lead to less work for the programmer. Even if the programmer expresses a suboptimal assignment such as Figure 5.7a the result will be Figure 5.7c when connection distances are minimized. This alleviates the programmer from agonizing over details that are better handled by an automated system.

It will also lead to better code. Instead of connecting cores at specific locations, programs are written to connect tasks. This is fundamentally different from the approach taken by workgroups where core coordinates are connected. By taking connectivity into account, much of the manual work associated with workgroups can be hidden and automated by the assignment algorithm. Most importantly, the order in which connections are created will not affect assignment.

To find a solution with specific properties, the assignment algorithm can try to minimize a cost function. Connection distance is easy to measure, and it is uniform in both directions. These properties makes connection distance a suitable metric to optimize when searching for a solution.

The cost function can be extended to take other network properties into consideration. Writing a cost function for connection distances is straightforward, but it might not give the best results. Other network properties such as limited bandwidth can also



(a) Naive solution **(b)** Close to fault **(c)** Minimum distance

Figure 5.8: Different assignment results when faults are considered.

be considered. Bandwidth becomes a problem when multiple cores shared the same network link (the same path through the mesh.)

The importance of good assignment results become more evident when the number of cores increase into the thousands. A naive implementation can displace a core from $(0, 0)$ to $(7, 7)$ in a 64-core chip, but in a 4096-core chip it can be displaced to $(64, 64)$.

5.2.2 Hardware considerations

In Figure 5.9 a faulty core is introduced at $(0, 2)$. This core is replaced by either $(3, 2)$ or $(0, 3)$, which solution to choose depends on the nature of the fault and hardware properties. The solution shown in Figure 5.9c is optimal in regards to connection distances. The two other solutions should work equally well on the Epiphany processor for some workloads. The solution in Figure 5.9a should produce minimal interference.

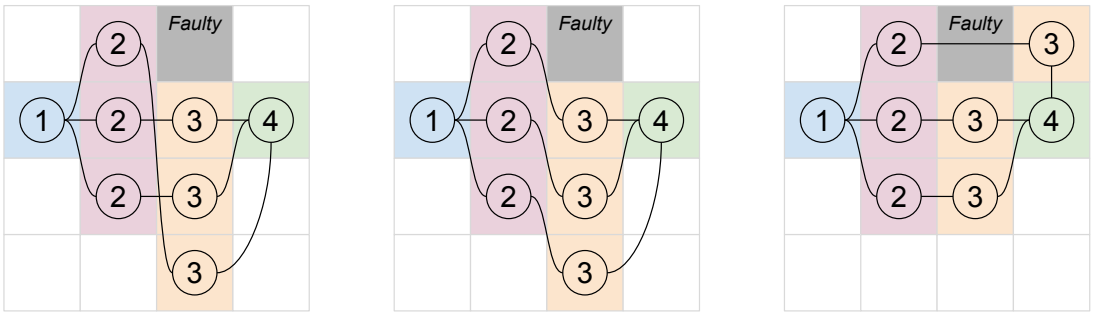
Simply optimizing connection distance might not produce the best results. As seen in Section 4.4 interference from neighboring cores and network congestion will affect performance. If correctly modeled by the cost function, these effects can be considered by the assignment algorithm.

5.2.3 Naive implementation

Two different assignment algorithms have been implemented: a naive algorithm and simulated annealing. This section will discuss the naive implementation. See Section 5.2.4 for more information about simulated annealing.

The naive implementation replaces each faulty core with a spare core; all other cores are left untouched: they are assigned their allocated task. The assignment matrix will therefore mirror the allocation matrix on fault-free chips.

Spare cores are chosen among all unallocated cores that fulfill the constraints. To avoid filling all loosely constrained cores first, the spare cores are ordered in ascending strictness.



(a) Minimal north-south interference

(b) No intersection

(c) Minimal distance

Figure 5.9: Different assignment results when faults are considered.

5.2.4 Simulated annealing

The optimizing assignment algorithm relies on simulated annealing to search for an optimal solution. This heuristic will find an approximate solution since finding an optimal is too expensive.

Simulated annealing is an algorithm for minimizing a cost function while searching for the global minima [16]. It lends its behavior from thermodynamics and cooling metal. When metal slowly cools its atoms fluctuate randomly and eventually arrive at a low energy state. Slow cooling and random fluctuations allow simulated annealing to escape local minima and find the global minima [16]. Simulated annealing has previously been used for routing wires, graph partitioning, solving the traveling salesman problem, and many others [17], [18].

Simulated annealing tries to minimize a cost function by iteratively exploring a design space [16]. The cost is allowed to increase and decrease while exploring, but the probability of accepting a costlier solution decreases for each iteration. This way simulated annealing can explore a large design space while eventually settling in the global minima, without getting stuck in local minimas.

Simulated annealing is easily adopted due to its iterative and random nature. In each iteration the location of two random tasks are swapped, the new cost is calculated, simulated annealing is applied, and eventually a good solution emerges. Implementing simulated annealing is straightforward and will not be covered here; however, the two most important concepts when mapping a many-core program will. Those concepts are: a cost function and a way of generating new solutions.

The cost function describes how optimal a solution is: a good solution has a low cost, and a bad solution has a high cost. The cost function used when mapping measures distances between all connected cores, and returns the sum of all Manhattan distances. Since the number of tasks and connections vary between applications so will the meaning of high and low cost. The connections are specified by a programmer or designer. All connections are assumed to have equal weight.

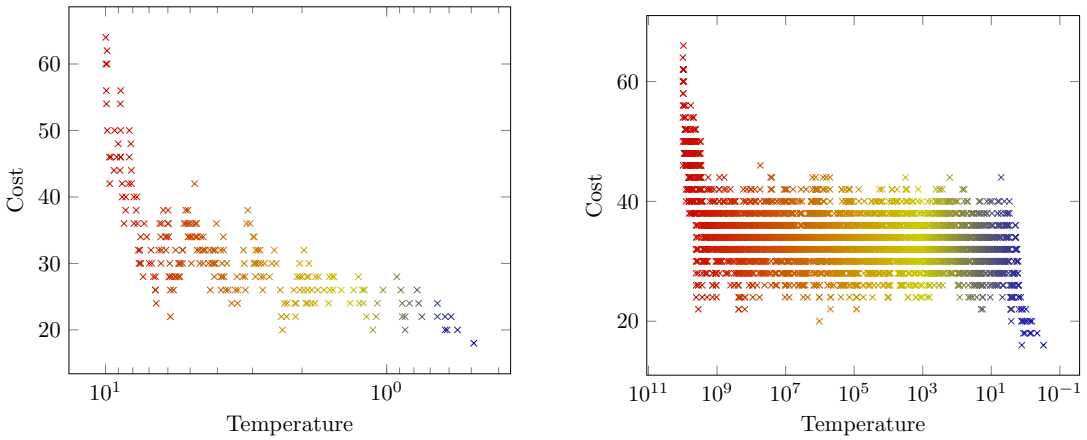
New solutions are generated by randomly swapping the task assignment of two cores: core A gets the task assigned to core B and core B gets the task assigned to core A. Spare cores, faulty cores, assigned cores and unused cores are all randomly swapped. Constraints are validated before the swap, and solutions that result in constraint violations are discarded. Assigning a task to a faulty core is considered a constraint violation. If no valid solutions exists the assignment algorithm returns an error code after testing a configurable number of invalid solutions.

Initially the algorithm uses a randomized solution. This solution is allowed to violated the design constraints. Constraints are resolved with each iteration and eventually all constraints are honored.

While the naive assignment algorithm only considers faulty cores, the optimizing algorithm considers all cores. Location data provided by the allocation matrix is discarded and all cores are moved.

Figure 5.10 shows the cost and temperature while mapping an example application with simulated annealing for two different starting temperatures. For each iteration the temperature decreases and so does the tendency to pick high-cost solutions. A low cost solution is eventually found for both starting temperatures; but the lower starting temperature in Figure 5.10a converges in 46 000 iterations while the higher starting temperature in Figure 5.10b requires 140 000 iterations.

Figure 5.10 illustrates the importance of picking a suitable starting temperature for simulated annealing. If the starting temperature is too high many iterations are wasted as in Figure 5.10b. High entropy results in large changes in cost for each iterations as illustrated in the temperature range 10^9–10^1. However, if the starting temperature is too low simulated annealing will settle in a local minima without finding the global minima. The suitable starting temperature will vary for different problems and different applications. It is therefore important to test different starting temperatures when mapping an



(a) Starting temperature: 10 **(b)** Starting temperature: 10^{10}

Figure 5.10: Temperature and cost during simulated annealing for two different starting temperatures.

application for the first time.

An incorrect stopping temperature will make simulated annealing abort prematurely or run unnecessary iterations. Premature aborts occur when the stopping temperature is too high and a good solution has not yet been found due to high entropy. Unnecessary iterations occur when the stopping temperature is too low and a good solution has already been found but no new solutions are accepted due to low entropy.

5.2.5 Partial remapping

The majority of tasks are not affected when a fault occurs, only a single core is affected. Therefore, remapping every task seems wasteful. Instead, we can perform a partial remap of tasks with faulty cores; keeping all other task assignments fixed. We call this procedure *partial remapping*.

Partial remapping takes existing task assignments and remap those tasks assigned to faulty cores. Therefore, only a single core is remapped if a single fault occurs. Tasks assigned to healthy cores are not moved. Thus, partial remapping will reuse most of the work performed when the application was previously mapped. This allows for an incremental remapping process.

Partial remapping requires that spare cores are reserved in a clever way. Simulated annealing packs all cores as tightly as possible; therefore, the closest spare core is far away when 4000 cores are packaged together. Partial remapping is useless when all cores a packed together: it will have no choice but to pick a spare core hundreds of cores away from its original location. To solve this issue, spare cores are added into the mix of healthy cores, or columns of spare cores are added at regular intervals. With this scheme the worst case distance to a spare core is controllable. Figure 5.11 shows two different patterns for reserving spare cores, with different amounts of redundancy.

	0	1	2	3	4
0		R			R
1		R			R
2		R			R
3		R			R
4		R			R

(a) Reserved columns

	0	1	2	3	4
0					
1		R			R
2					
3					
4		R			R

(b) Reserved islands

Figure 5.11: Different patterns of reserved cores for use with partial remapping. Reserved cores are denoted with *R*.

5.3 Results

The time complexity of the implemented naive assignment algorithm is shown in (5.1), where N is the number of cores, F the number of faults, and C the number of constraints. The complexity is proportional to FC^2 due to a sorting step with complexity $\mathcal{O}\left(C^2\right)$. The sort complexity can be reduced to $\mathcal{O}\left(C \log C\right)$ by using another sorting algorithm.

$$\mathcal{O}\left(4N + F(N + C + C^2)\right) \tag{5.1}$$

The maximum number of simulated annealing iterations are shown in (5.2), where T_s is the stop temperature, T_0 the start temperature, and c the cooling factor. The factor 2 was added to consider discarded iterations that violate design constraints, in the worst case. In reality this factor spans from 1 to 2.

$$2 \left\lceil \frac{\log T_s - \log T_0}{\log c} \right\rceil \tag{5.2}$$

Time complexity analysis does not apply to heuristics, but it can be interesting to consider anyway. The time complexity of the implemented simulated annealing solver is shown in (5.3), where N is the number of cores, L the number of links, k the number of iterations from (5.2), and q is the number of found solutions with a lower cost.

$$\mathcal{O}\left(8N + qN + L + kL\right) \tag{5.3}$$

As shown by (5.3) the time complexity for mapping an application increases linearly with the number of cores N. Figure 5.12 shows the expected linear increase in mapping time when the same application is mapped for different numbers of cores. The same start and stop temperature was used for all iterations. The mapped application forms a chain where each core communicates with its neighbor. All cores are allocated, and form an n by n square; no spares exist. It will take 40 minutes to map 4000 cores on Parallella's ARM processor if the trend holds.

Figure 5.13 shows the results of mapping the pipeline application with four different fault matrices. Figure 5.13a shows a mapping without faults while Figure 5.13b and Figure 5.13c shows one and two faults respectively. Figure 5.13d shows four faults. The cost of introducing a single fault equals the cost of no faults; however, cost increases when two or more faults are introduced. Each solutions targets a 64-core chip; the figures only shows the top left corner of 4 by 4 cores.

Figure 5.14 shows the results of partially remapping the pipeline application. Figure 5.14a shows the initial fault free mapping and the cores that where reserved during this phase. Figure 5.14b shows the result of partially remapping the application with four faults; only two tasks are reassigned.

5.4 Discussion

The random nature of simulated annealing becomes evident when comparing the cost of Figure 5.13c and Figure 5.13d. The first solution with two faults results in a higher cost

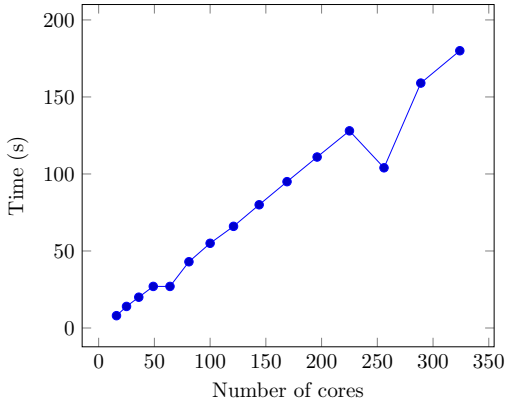
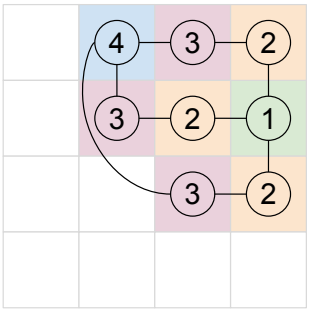
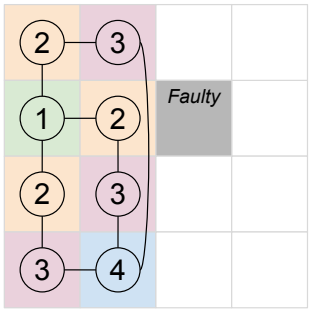


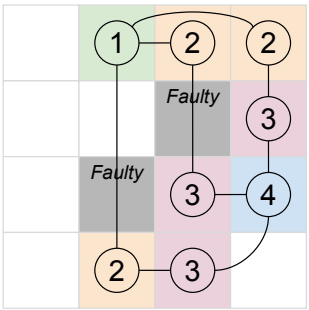
Figure 5.12: Mapping time for different number of cores but same application.



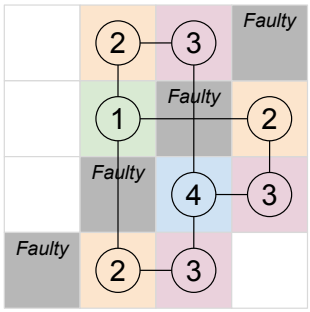
(a) No faults (cost: 11)



(b) One fault (cost: 11)

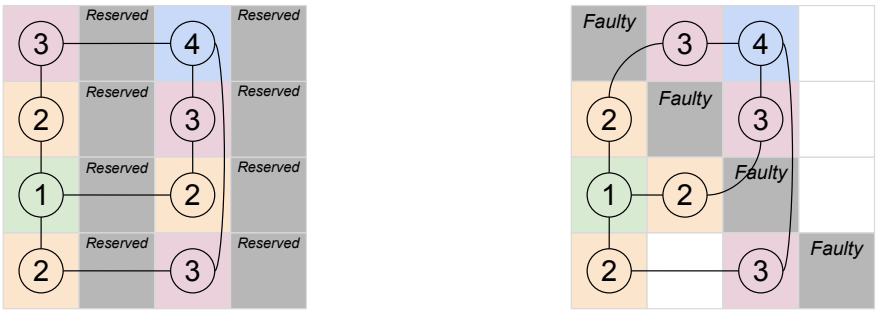


(c) Two faults (cost: 14)



(d) Four faults (cost: 12)

Figure 5.13: Results of mapping the pipeline application with simulated annealing.



(a) No faults (cost: 14) **(b)** Four faults (cost: 14)

Figure 5.14: Results of partially remapping the pipeline application with simulated annealing.

than the second solution with four faults. Since simulated annealing is a heuristic it will produce different solutions each run. If Figure 5.13c is remapped with a different seed it will produce a different solution and possibly a lower cost. Simulated annealing makes a trade-off between run time, implementation difficulty, and quality. These solutions took under a second to produce and the implementation is 200 lines of C-code.

For each application there is a minimum cost. Two cores cannot be closer than one distance unit, and all distances are integers (unless links are weighted to consider usage.) Simulated annealing could abort when this cost is reached, or when the cost is close to its minimum value. This will reduce the number of iterations and speed up assignment. However, finding the minimum cost is difficult, and it will vary between applications. It could be found by running simulated annealing multiple times with the same configuration and application.

Every core is remapped with simulated annealing and information that could potentially reduce mapping time is discarded. No information is retained from previous assignments or the allocation matrix. When faults occur the mapping is still partially valid since only a limited number of faults occur simultaneously. Exploiting this fact by only remapping affected cores could improve mapping performance. But a partial remapping will limit the design space and reduce the chance for simulated annealing to find an optimal solution. However, simulated annealing requires an initial solution, which is currently randomized. Using the previous mapping as an initial solution could improve mapping performance if a suitable starting temperature is chosen. More on partial remapping later.

The trend shown in Figure 5.12 predicts that it will take 40 minutes to map 4000 cores. 40 minutes is a long downtime for real-time applications, but it is considerably shorter than the time it takes to replace a faulty processor. The mapping time can be reduced by running the mapper on a faster processor (faster than an ARM Cortex A9), and by applying clever optimizations such as partial and hierarchical mapping.

The dips in mapping time shown in Figure 5.12 occur at 64 and 256 cores. For these two configurations the mapping time is lower than expected. The same pattern will

probably occur at 512, 1024, 2048, and for other values of N where $N = 2^n$. Since the mapping system was recompiled between runs, the compiler was able to better optimize the code for these cases. Therefore, it will likely take less than 40 minutes to map 4096 cores.

Partial remapping requires that spare cores are reserved when the application is first mapped. When faults occur these reserved cores are used as spares, but under fault free conditions they are unused. This will introduce gaps of unused cores, gaps that increase the distance between tasks. Choosing how many cores to reserve, and in which pattern, will therefore affect the performance of fault-free chips. Reserving columns as in Figure 5.11a is well suited for partial remapping, but it results in 40 % redundancy, and therefore many unused cores. Reserving islands as in Figure 5.11b results in 16 % redundancy, but partial remapping will only produce good results if a single faults occur in the same region of 3 by 3 cores. A 3 by 3 region only contains a single spare core, a second fault in the same region will use a spare core far away from its original location.

From (5.3) and Figure 5.12 we know that mapping time increases linearly with the number of cores. We also know that it depends on the number of links and communicating tasks. This is also true for partial remapping. Full and partial remapping use the same cost function, the same links and the same allocation matrix. Since evaluating the cost function is the most expensive operation in each iteration, performing a partial remap is almost as time consuming as performing a full remap, if the same number of iterations are executed. A carefully chosen starting temperature, stopping temperature, and cooling rate is vital to reduce the number of iterations. Making a good choice is difficult, especially when configuring for unknown scenarios. However, we do know that partial remapping is most likely invoked with a single faulty core and many spares. The configuration should therefore use a small temperature range and a large cooling factor (when compared to full remapping) to efficiently explore the limited design space.

Instead of mapping all cores as a large coherent graph, each functional unit can be mapped separately in a hierarchical manner. Redundant cores are added to each functional unit to form a square or rectangular block. All rectangular blocks are then mapped in a separate stage. This should reduce mapping time and introduce spare cores into each functional block, allowing partial remapping to remap each block separately.

Pre-computing can reduce downtime when remapping. During compile time, or while the host processor is idle, we can compute all possible maps with a single faulty core, and store those on disk. For a 4096-core processor this will yield 4096 files, with a total size of 16 MiB (one byte per core.) When a fault occurs we simply load the pre-computed map that ignores the affected core. With this approach remapping time is only limited by the time it takes to read the map from disk and restart the user application (if we ignore time spent diagnosing the hardware.) If the pre-computed maps are created during compile time this speedup is only applicable to the first fault. However, if the pre-computed maps are created on the host processor, new maps can be created each time a fault occurs.

The cost function only accounts for distances but can be extended to include router usage and other network phenomena. Distance is not necessarily the most important

metric; the number of cores that send data through the same router, and the behavior of the network arbiter, will also impact network performance. Extending the cost function to account for these effects will produce better mapping results. However, modeling these effects correctly, and weighing them in relation to each-other, is challenging. The impact of these effects depend on the communication patterns of the application — two cores that communicate at different times do not interfere — which complicates matters even more. Because of these complications, it was decided to keep the cost function as simple as possible, and to only account for distance. Complicated cost functions will degrade remapping performance if they are computationally expensive to evaluate.

Extending the cost function to account for the amount of data sent between cores is achieved by added weights to each connection. The weights are arbitrary floating point numbers that correspond to the amount and frequency of sent data. Their absolute values are not important, only the relations between different weights matter. The distance between cores is multiplied by the weight, which will results in shorter heavy connections and longer light connections.

The cost function does not consider the distance between the host and the core. Data sent from the host processor to a core will traverse the network. This data is sent on a separate mesh, and will therefore, not interfere with inter-core communications. However, the distance between the core and the host-device network interface will affect communication latencies. Data sent from the host enters the device along the left most column. Cores that receive input data should be placed along this column to reduce latency. Host-device communications are seldom performance critical, and were therefore ignored. However, the cost function can be modified to take it into consideration.

The simulated annealing mapper can be applied to projects that do not target fault tolerance. Manually assigning tasks is a daunting process which the mapper can automate. In its current form the mapper produces a new mapping each time an application starts. This behavior is not suitable for use with non-fault tolerant projects. These projects are better served by a static mapping that is generated once, and generated off-line. For this purpose the mapper can be rewritten as a standalone tool. This tool will read allocation and constraint matrices from files, generate a static assignment matrix, and write the results to disk. The result can be stored as XML, as C-header files, or as simple text files. The static mapping is read by the application at startup.

6

Conclusion

This report presents two different remapping strategies: workgroup remapping and matrix remapping. These strategies are fundamentally different in their API and resulting mapping. Workgroup remapping is useful for existing applications targeting the *Epiphany software development kit* (ESDK). These applications rely on few cores (existing hardware supports up to 16-cores) and workgroup remapping is easily adopted due to its similarities with the ESDK. More importantly, workgroup remapping is useful when targeting hardware without fabrication defects. When faults occur it will quickly remap the application, keeping it running until the hardware is replaced, but with reduced performance. Matrix remapping is better suited for new projects and applications that are written from the ground up with remapping in mind. It is well suited for hardware with fabrication defects since it will map the application with minimum performance penalty.

Remapping is suitable for tolerating permanent hardware faults, but not transient faults. Rebooting is a lengthy process, and so is hardware diagnosis and remapping. The downtime can be measured in minutes since the system must reboot, rerun hardware diagnosis, and then remap itself upon faults. Remapping alone can take between 1 second to 40 minutes depending on the application and the number of cores. Remapping should therefore target permanent hardware faults, such as wear-out faults and manufacturing faults, where rebooting and remapping is a small price to pay compared to replacing the hardware. Intermittent and transient faults are better tolerated by error correcting codes and other hardware techniques. Remapping avoids faulty cores, nothing else. No attempts are made to rescue the application state when faults are detected. This behavior is acceptable for some applications but limits the usefulness of remapping when used as the only fault tolerance method. However, applications are free to implement other fault tolerance methods that complement remapping.

Simulated annealing is a good fit for mapping parallel applications on the Epiphany architecture. Cost functions can model different behaviors and consider optimizations that are otherwise difficult to implement. Mapping with simulated annealing is useful

for projects not concerned with fault tolerance. It allows programmers and designers to describe their application, and connections between tasks, without considering task assignment. This will reduce program complexity and increase maintainability. The need for nested for-loops and meticulous mapping disappears. It can also increase performance by utilizing optimization opportunities overlooked weary programmers.

It is currently unknown if faulty processors will be sold in the future, and if selling faulty hardware will increase revenue or merely result in usability issues. Faulty cores in the Cell processor were masked using built-in hardware fault tolerance. Future processors might take a similar approach or rely on software methods such as remapping. Only time will tell.

7

Future work

Remapping is a broad topic and the solutions presented here are by no means complete. There is much more work to be done. This chapter introduces some of that work.

7.1 Automated relations

When writing many-core software for the Epiphany platform a significant amount of work and code is dedicated to finding and computing the addresses to shared memory. For example, in a pipelined application each stage of the pipeline needs a pointer to the next stage. Shared memory addresses are currently computed by converting a core ID into a global memory address and by adding an offset. For each stage a special software routing is written to compute the addresses and to set up pointers. Writing these routines manually is error prone and labor intensive; furthermore, it does not scale to large applications with many different tasks. Finding an automatic or semi-automatic way of computing the addresses would greatly simplify the process of writing relocatable software. For example, the relations between tasks could be expressed using a domain specific language.

7.2 RTL analysis

An analysis of the Epiphany core RTL-code could identify issues with fault containment and evaluate the credibility of a remapping system. If the Epiphany RTL-code is unavailable, OpenRISC or similar processors can be used instead. By injecting stuck-at-faults into the RTL it should be possible to investigate the propagation of such faults. Meixner and Sorin [12] performed similar research on processors from undisclosed vendors.

7.3 Assembly code analysis

To run programs on partially faulty cores an analysis of the program assembly code is required. This analysis will determine which parts of the processor are required for correct execution. If a device program does not use floating point instructions it could run on cores with faults in the floating point unit. Similarly, programs that do not use memory bank 3 could run on cores with faults in memory bank 3.

7.4 Extended cost function

The cost function used when mapping with simulated annealing can be extended to model the hardware more closely. A more accurate cost function could increase application performance by better utilizing the mesh. It could also hurt mapping performance, or turn out to be superfluous.

7.5 Compare genetic algorithms

Simulated annealing is an old algorithm. Many modern contenders exist, and it would be interesting to compare these with the mapping results achieved by simulated annealing. It would also be interesting to evaluate hierarchical and partial remapping.

Bibliography

[1] Adapteva Inc. Epiphany architecture reference manual.

[2] A. Varghese, B. Edwards, G. Mitra, and A. P. Rendell, "Programming the adapteva epiphany 64-core network-on-chip coprocessor," *CoRR*, vol. abs/1410.8772, 2014. [Online]. Available: http://arxiv.org/abs/1410.8772

[3] A. Inc, "Parallella-1.x reference mandual," Adapteva Inc, 2014, access March 2015. [Online]. Available: http://www.parallella.org/docs/parallella_manual.pdf

[4] ——, "Epiphany SDK reference," Adapteva Inc, 2013, access December 2014. [Online]. Available: http://adaptev.com/docs/epiphany_sdk_ref.pdf

[5] B. Johnson, *The Electrical Engineering Handbook*, 2nd ed. CRC Press, 1997, ch. Fault Tolerance, p. 2171.

[6] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1, pp. 11–33, Jan 2004.

[7] S. Muller, M. Scholzel, and H. Vierhaus, "Towards a graceful degradable multicore-system by hierarchical handling of hard errors," in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, Feb 2013, pp. 302–309.

[8] M. Breuer, S. Gupta, and T. Mak, "Defect and error tolerance in the presence of massive numbers of defects," *Design Test of Computers, IEEE*, vol. 21, no. 3, pp. 216–227, May 2004.

[9] R. M. Pathan, "Three aspects of real-time multiprocessor scheduling: Timeliness, fault tolerance, mixed criticality," Ph.D. dissertation, Chalmers University of Technology, 2012.

[10] M. Andersson, "Integrated circuit yield enhancement," Master's thesis, Chalmers University of Technology, 2010.

[11] B. F. Romanescu and D. J. Sorin, "Core cannibalization architecture: Improving lifetime chip performance for multicore processors in the presence of hard faults," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08, 2008, pp. 43–51.

[12] A. Meixner and D. Sorin, "Detouring: Translating software to circumvent hard faults in simple cores," in *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, June 2008, pp. 80–89.

[13] J. Schofield, "Could 10 - 20 % yields for Cell processors lead to problems for Sony PS3?" The Guardian, July 2006, accessed February 2015. [Online]. Available: http://www.theguardian.com/technology/blog/2006/jul/15/could1020yie

[14] M. Stan and P. Re, "Electromigration-aware design," in *Circuit Theory and Design, 2009. ECCTD 2009. European Conference on*, Aug 2009, pp. 786–789.

[15] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "Swift: software implemented fault tolerance," in *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, March 2005, pp. 243–254.

[16] W. L. Goffe, G. D. Ferrier, and J. Rogers, "Global optimization of statistical functions with simulated annealing," *Journal of Econometrics*, vol. 60, no. 1, pp. 65–99, 1994.

[17] L. Ingber, "Very fast simulated re-annealing," *Mathematical and computer modelling*, vol. 12, no. 8, pp. 967–973, 1989.

[18] T. Bertsimas, "Simulated annealing," *Statistical Science*, vol. 8, no. 1, pp. 10–15, 1993.