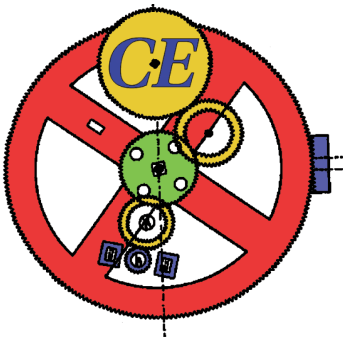


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2018

MSc THESIS

Automated Multi-core Scheduling for an Industrial-sized Mechatronic Motion Control Platform

D.N.F. Brouwer

Abstract

Mechatronic embedded control systems are becoming increasingly sophisticated and computationally demanding. These systems typically consists of multiple controllers, which coordinate the actuators and apply feedback based on data collected by sensors. Often the underlying control strategy is entirely described in a software application, which allows for hardware independence and adds the ability to conveniently change algorithms. In order to increase application throughput, a commonly used approach is to divide the application into smaller units called *tasks* and execute them in parallel using multi-core hardware. In this thesis an automated multi-core aware scheduling and assignation approach is designed for an industrial-sized mechatronic control software platform, more specific the Prodrive Motion Platform (PMP). PMP can be applied in a wide range of products, e.g. wafer scanners, robots, elevators. A key feature of PMP is flexibility, which allows it to be utilized in combination with a wide variety of both controllable hardware (actuators, sensors) as well as computational hardware. As a direct consequence, PMP supports many different customers and corresponding requirements. Within PMP, a customer typically defines the application, which is then translated into a set of tasks. This task-set is then scheduled and assigned onto the available multi-core hardware resources. In order to meet timing-constraints, the current scheduling approach relies on a time-consuming manual process, which provides a limited amount of tuning options, and neither considers *task* workloads, nor inter-core communication costs. Given these short-comings, this solution proved to be unsuited for upcoming PMP products. To address the aforementioned issues, we first review state of the art scheduling solutions and introduce an extensible task measurement framework. Subsequently, we evaluate various scheduling approaches on current PMP applications and identify two algorithms, namely, *Internalization using Load Balancing* and *DCS*, that are able to automatically find schedules, whilst still meeting application timing-constraints. Besides enabling to schedule new applications within upcoming PMP products, performance improvements of $\sim 3.3\%$ and $\sim 2.0\%$ were observed by *Internalization using Load Balancing* and *DCS*, respectively, compared to the original scheduling approach within the multi-core PMP product PPCx3. Last but not least it is shown that within a relatively new product XEONx3, our approach provides performance improvements of 34.30% up to 49.61%, depending on the utilized scheduling algorithm.

CE-MS-2018-28

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Automated Multi-core Scheduling for an Industrial-sized Mechatronic Motion Control Platform

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

specialized in

COMPUTER ARCHITECTURE

by

D.N.F. Brouwer
born in Rotterdam, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Automated Multi-core Scheduling for an Industrial-sized Mechatronic Motion Control Platform

by D.N.F. Brouwer

Abstract

Mechatronic embedded control systems are becoming increasingly sophisticated and computationally demanding. These systems typically consists of multiple controllers, which coordinate the actuators and apply feedback based on data collected by sensors. Often the underlying control strategy is entirely described in a software application, which allows for hardware independence and adds the ability to conveniently change algorithms. In order to increase application throughput, a commonly used approach is to divide the application into smaller units called *tasks* and execute them in parallel using multi-core hardware. In this thesis an automated multi-core aware scheduling and assignation approach is designed for an industrial-sized mechatronic control software platform, more specific the Prodrive Motion Platform (PMP). PMP can be applied in a wide range of products, e.g. wafer scanners, robots, elevators. A key feature of PMP is flexibility, which allows it to be utilized in combination with a wide variety of both controllable hardware (actuators, sensors) as well as computational hardware. As a direct consequence, PMP supports many different customers and corresponding requirements. Within PMP, a customer typically defines the application, which is then translated into a set of tasks. This task-set is then scheduled and assigned onto the available multi-core hardware resources. In order to meet timing-constraints, the current scheduling approach relies on a time-consuming manual process, which provides a limited amount of tuning options, and neither considers *task* workloads, nor inter-core communication costs. Given these short-comings, this solution proved to be unsuited for upcoming PMP products. To address the aforementioned issues, we first review state of the art scheduling solutions and introduce an extensible task measurement framework. Subsequently, we evaluate various scheduling approaches on current PMP applications and identify two algorithms, namely, *Internalization using Load Balancing* and *DCS*, that are able to automatically find schedules, whilst still meeting application timing-constraints. Besides enabling to schedule new applications within upcoming PMP products, performance improvements of $\sim 3.3\%$ and $\sim 2.0\%$ were observed by *Internalization using Load Balancing* and *DCS*, respectively, compared to the original scheduling approach within the multi-core PMP product PPCx3. Last but not least it is shown that within a relatively new product XEONx3, our approach provides performance improvements of 34.30% up to 49.61%, depending on the utilized scheduling algorithm.

Laboratory : Computer Engineering
Codenumber : CE-MS-2018-28
Committee Members :

Advisor:	Dr. S.D. Cotofana, CE, TU Delft
Chairperson:	Dr. S.D. Cotofana, CE, TU Delft
Member:	Dr. T.G.R.M. van Leuken, CE, TU Delft
Member:	J.J.A. Kuijsten M.Sc., PMP, Prodrive Technologies

Dedicated to my family.

Contents

List of Figures

List of Tables

List of Acronyms

Acronyms

API	Application Programming Interface.
BCET	Best Case Execution Time.
BNP	Bounded Number of Processors.
CLC	Control Loop Component.
CP	Critical Path.
DAG	Directed Acyclic Graph.
DS	Dominant Sequence.
EFT	Earliest Finishing Time.
ESL	Extendable Scheduler Library.
EST	Earliest Start Time.
MMU	Memory Management Unit.
NRTC	Non Real-Time Controller.
PA	Processor Assignment.
PMP	Prodrive Motion Platform.
PT	Parallel Time.
RTC	Real-Time Controller.
ST	Sequential Time.
UNP	Unbounded Number of Processors.
WCC	Worst Case Cycle.
WCET	Worst Case Execution Time.
WSS	Working Set Size.
XML	Extensible Markup Language.

Acknowledgements

I would like to thank my family and friends, who have always supported and encouraged me throughout my study and master's research project. In addition, I want to thank all of my colleagues and new friends at Prodrive, who have helped me finish my research project and gave me all the freedom in finding a solution for their amazing motion platform. Special thanks to my supervisor Jasper Kuijsten M.Sc. who devoted his time and effort to support me whenever I required it. Last but not least, special thanks to Dr. Sorin Cotofana for enabling me to do my research and supporting me all the way through.

D.N.F. Brouwer
Delft, The Netherlands
December 11, 2018

1 Introduction

Industrial-sized mechatronic control systems are becoming increasingly sophisticated. Many of these systems consist of multiple controllable components, which all need to be managed and monitored for the entire system to function properly. Consider a robot inside a factory. In order to assemble a product, such a robot requires a certain freedom of movement. These movements are carried out by a collection of axes, each of them allowing for an additional degree of freedom. All these axes require both individual and combined coordination, in order to perform the desired movements.

In order for the robot to move along on of its axes, actuators are required. Example actuators are: servos, linear motors, and solenoids. These actuators are typically hydraulic, pneumatic, or electric powered. Irrespective of the power source, most actuators are controlled in the electrical domain by means of electric signals. In order to allow for feedback, there is also a need for sensors. Example sensors are: encoders, strain gauges, and accelerometers. Sensors come in many forms and applicable domains, though similar to the actuators, most sensors eventually convert their signals to the electrical domain. In order to coordinate the actuators and apply feedback using the signals supplied by the sensors, a controller is required. Besides generating the required electric signals, a controller typically implements a control algorithm (e.g. a PID controller). Often these algorithms are described in software, which allows for hardware independence, moreover adds the ability to conveniently change the applied algorithm.

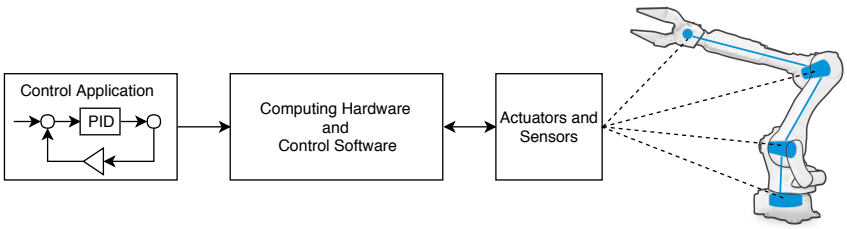


Figure 1.1: Controlling and managing a mechatronic control system.

Typically an entire mechatronic system -like the robot example- consists of multiple controllers and therefore multiple software descriptions of control algorithms. The complete set of software descriptions, required to manage and control a mechatronic device, is known as an application. Figure 1.1 presents a high-level overview of all the components required for controlling a mechatronic system. In order for the robot to (safely) assemble a product, a certain throughput is required for the application. In the motion control domain, this throughput requirement is often expressed in terms of the desired control frequency, which relates to the rate at which each control algorithm needs to be executed.

Due to time criticality, these systems are often classified as being real-time and depending on the application and safety aspects they reside between soft and hard real-time.

In order to increase application throughput, a commonly used approach is to parallelize the computations by dividing the application into smaller pieces called *tasks*, which can then be executed onto multi-core hardware. This process typically involves: mapping tasks onto hardware resources (assignation) and determining the proper task execution order (scheduling), which both have to consider task dependencies and ordering, in order to preserve the targeted application behaviour. Since manual task assignation and scheduling is a complex and time consuming process, moreover industrial-sized mechatronic applications are becoming increasingly sophisticated, mechatronic platform providers are interested in migrating to automated solutions.

In this line of reasoning we investigate and introduce in this thesis an automated multi-core aware scheduling and assignation approach for an industrial-sized mechatronic embedded control platform called the Prodrive Motion Platform (PMP). More specifically we seek a replacement for the current time-consuming approach, able to at least match the performance of the current solution, without requiring any human guidance or manual tweaking.

Multi-core scheduling in general is a well known NP Complete problem [1], with the exception of a small sub-set of specific (theoretical) configurations [2], [3], [4]. These exceptional cases rely on specific assumptions, e.g. communication-less tasks, strict bound on the amount of cores, preemption, variation-less task timings, which unfortunately do not hold in most practical systems, PMP included.

1.1 Problem Statement

PMP is a generic collection of real-time and non-real-time motion control software, that can be used in a wide range of mechatronic products, e.g. wafer-scanners, robots, elevators. A hardware product that implements PMP software, is known as a PMP product. Besides the necessary software, Prodrive also supplies hardware, which includes controllable hardware (drives, sensors, and actuators), and computing hardware. Due to the wide variety in customers and corresponding requirements, both the controllable, as well as the computing hardware (responsible for application scheduling and execution), differs per product. To support the wide variety in mechanical and computation hardware, the PMP software is designed in a generic fashion.

Within the PMP software, the customer's application is translated into a task-set. Due to the wide variety of available hardware resources and the possibility for customers to implement their own control algorithms, there is a wide-variety in task-set structures. Furthermore, the workload of each task is unpredictable in general.

Up to date, a certain line of actions has been taken to optimize the application throughput, in order to meet the timing requirements for current PMP products. Multi-core hardware support has been added and a multi-core aware assignation method has been implemented in the PMP software. In Figure 1.2 an example task-set corresponding to a customer's application, is presented; in which tasks are represented by nodes and dependencies are represented by edges.

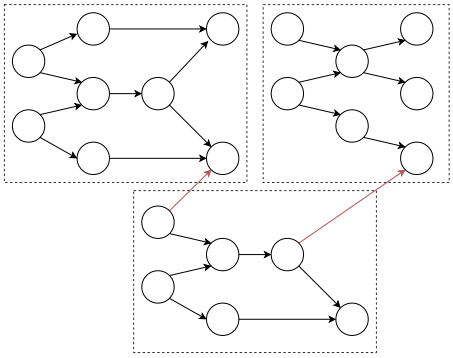


Figure 1.2: An example task-set.

Nodes within a rectangle represent tasks belonging to the same controller.

As portrayed in Figure 1.2, the amount of dependencies existing between tasks belonging to a specific controller, is generally higher than the amount of dependencies between controllers. In this context a controller is a collection of sensor, actuator, and control algorithm tasks, required to perform control operations on some mechatronic sub-system (e.g. control a single linear stage in a wafer-stepper).

The initial assignation approach (which is currently used as a fall-back method), is based on a heuristic that tries to evenly spread the assignation of controllers to the available cores. This approach relies on the reoccurring structures seen in past and current PMP products (as portrayed in Figure 1.2), thereby trying to reduce the amount of core-to-core communications. However, since the heuristic ignores the underlying graph structure and does not take the computational weight of each task into account, it fails to meet the application throughput requirements for most PMP products. In order to solve this problem, a second assignation approach was implemented that relies on manually optimized static configuration files, that specify controller to core mappings.

Using a manually optimized configuration, throughput constraints are met for current products, however, the process of constructing a configuration is not in line with the generic design philosophy of the platform. Each new or modified application, requires manual (re-)tuning in order to meet the customer application throughput requirements. Given that the manual tuning options are limited and the fact that execution time measurements are performed on a complete application execution cycle, rather than at task-level granularity, the manual tuning process is sub-optimal, time-consuming, and mainly based on a trial-and-error approach. In addition, due to the increase in complexity of PMP products, it is expected that the manual approach will not be practically feasible anymore within the near future of the platform.

In order to solve the problems at hand and allow for future developments within the platform, the following research question will be addressed within this thesis:

Which automated scheduling and assignation approach would be a worthy replacement for the current time-consuming sub-optimal approach?

1.2 Requirements and Constraints

To address the main research question, it should first be defined what determines the worthiness of a new approach as a replacement for the current approach. In addition to eliminating the need for human intervention, the automated approach must adhere to the following performance requirements:

- In terms of application execution time, the proposed scheduling and assignation approach should:
  - Outperform the current approach without applying any manual optimizations (i.e. using the heuristic).
  - Perform at least equally well whilst applying a configuration that has been manually optimized in a time-consuming process.
- The additional initialization time of the system due to scheduling operations should, on average, be less than a minute.

1.3 Contributions

In order to answer the research question, several analysis and design steps have been taken. Since PMP is of substantial size, as a first step within the analysis, we created an abstract model of the system that only contains the relevant parts with regards to the scheduling and assignation of a customer's application. Then the current approach was analyzed in order to get more insight as to why this approach was chosen, moreover find out the requirements for an automated alternative. With these requirements in mind, state of the art multi-core scheduling algorithms were investigated that would be applicable within PMP.

In order to set a baseline, the performance of typical control applications on two multi-core based PMP products, namely PPCx3 and XEONx3, were analyzed whilst using the current scheduling and assignation approach. The resulting performance was used in combination with the system and state of the art analyses, to come up with a design strategy.

PMP is a relatively large software platform that is actively maintained and extended by a team of more than thirty engineers. In order to maintain stability, testing is of great importance. In the search for an automated scheduling and assignation solution, we opted for an iterative design strategy, which allowed for intermediate testing. The design was divided into two main parts: a measuring, and a scheduling framework. Based on intermediate test results, we decided upon the next iterative design step. In the end, a measuring framework was implemented, which, using a number of different measurement strategies, is able to determine the computational weight of each task and the overhead during core-to-core communication. In addition, a generic scheduling framework was implemented, which includes a library of scheduling algorithms able to use the measurements from the measuring framework, in order to perform assignation and scheduling in an automated way.

In total five different scheduling and assignation implementations have been added to the library, which includes three algorithms from the state-of-the-art analysis, i.e. *HLFET* [5], *ISH* [6], and *Internalization* with *Sarkar*'s Processor Assignment [7]. Since neither of the algorithms could satisfy all performance requirements, a *Load Balancing* based PA extension for the *Internalization* algorithm is presented, which in a typical control application on the PowerPC (3x e500) based wafer-stepper product PPCx3, outperforms the original scheduling approach whilst using a manually optimized schedule by $\sim 3.3\%$. Without performing the manual optimization step, thus falling back to the heuristic, a performance increase of $\sim 5.2\%$ was observed.

Besides the *Load Balancing* PA extension, a clustering based, iterative scheduling approach named *DCS* is proposed, which unlike the algorithms from the state-of-the-art, takes into account independent sub-graph structures (which are common within PPCx3), moreover uses the measuring framework to improve upon execution time estimations. In PPCx3, the *DCS* algorithm is able to perform $\sim 2.0\%$ better using the manually optimized variant and $\sim 3.8\%$ better than the heuristic.

Last but not least it is shown that for a typical control application within the relatively new, Xeon based (3x D-1500) wafer-inspection machine XEONx3, in which the manual optimization process is not possible (yet) and the fall-back heuristic is used as the default scheduler, a performance improvement of 34.30% up to 49.61% can be observed, depending on the scheduler that is chosen from the library.

1.4 Organization

In Chapter 2 a general overview of the Prodrive Motion Platform is presented, which includes a description of the current assignation and scheduling solution, whilst also presenting its shortcomings and advantages. Chapter 3 includes a presentation of all relevant concepts concerning (multi-core) scheduling and assignation. In addition, an overview of state-of-the-art scheduling and assignation literature is presented. Chapter 4 provides an in-depth performance analysis of the scheduling and assignation solution, for current multi-core PMP products. Chapter 5 includes the design of an automated scheduling and assignation solution. In Chapter 6 all relevant steps regarding the implementation are presented, including all intermediate results; the chapter ends with a comparison and scalability analysis. Last but not least, in Chapter 7 the conclusions are formed, followed by a discussion which includes a future work presentation.

2 The Prodrive Motion Platform

In this chapter, a general overview of the Prodrive Motion Platform (PMP) is presented. PMP is a generic collection of real-time and non-real-time motion control software, that can be used in a wide range of mechatronic products, e.g. wafer-scanners, robots, elevators. Typically a customer requires hardware and software for their mechatronic product (e.g. a robotic arm). Such a product may consist of multiple servos (actuators) and encoders (sensors). Besides the PMP software, Prodrive produces and provides all necessary hardware, ranging from sensors and actuators, to power electronics, industrial cabinets and high-end computing platforms. In addition to the hardware, a customer requires an Application Programming Interface (API) in order to interface with the hardware, and deploy their own application.

Whereas PMP refers to the complete collection of generic motion software, a PMP product is defined as an entire system consisting of all necessary actuators, sensors, cabinets, power electronics and computing hardware, in combination with all the required software for the product to function.

Since not all hardware and software components within PMP are relevant with regards to the scheduling and assignation problem, a simplified model representing a PMP product, is presented. Figure 2.1 depicts an abstract model for the hardware within a PMP product. Within this model a PMP product consists of: a master controller, drives, sensors and actuators.

PMP drives are used as an interface for all the sensors and actuators within the system. These drives consists of power electronics and may contain computing hardware (FPGAs, DSPs, etc.), used for pre-processing and post-processing, sensor and actuator values. The amount and type of drives, actuators, and sensors; is product specific. Figure 2.1 depicts an example robotic arm, consisting of multiple sensors and actors, which are connected via the drives, to the master controller.

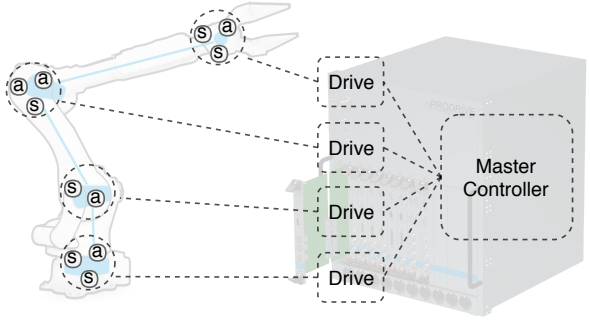


Figure 2.1: (simplified) PMP hardware model.

2.1 The Master Controller

The master controller is the centerpiece of the motion platform. A master controller consist of (multi-core) computing hardware in combination with PMP software. Among other things, the master controller is used to process the customer application. Due to the wide variety in customer requirements, the computing hardware varies per product, i.e. the architecture and amount of available cores, is product specific. In order to support this wide variety in hardware, the software executing on the master controller, is designed in a generic fashion.

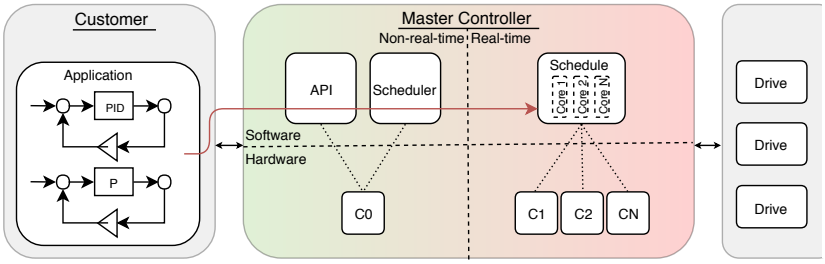


Figure 2.2: Hardware and software model for the master controller.

The red arrow shows the flow for scheduling a customer's application onto the real-time hardware resources.

In Figure 2.2 an abstract model is presented for the master controller, which includes both the software and hardware. Within the master controller, one of the available cores is reserved for all non-real-time computations which includes, scheduling, preparation of the real-time system, and providing an interface with the customer. As presented in Figure 2.2, a customer is able to upload their application description to the master controller through the API. After uploading, a customer is able to request for real-time execution of the supplied application, whereafter a schedule and assignation process is initiated on the non-real-time part of the master controller. In this process, the application description of the customer is transformed into a task-graph consisting of tasks and dependencies, after which each task is assigned to one of the real-time cores. In addition, a task execution order (schedule) is created. When both the assignation and scheduling processes have finished, the real-time portion of the master controller starts executing accordingly. During execution, a customer is unable to alter their application description. If changes are required, a customer is able to halt execution of the current application, after which the customer can run through the same process again, to upload their updated application.

The software running on the non-real time core is referred to as the Non Real-Time Controller (NRTC), in which -as the name implies- all non-real-time related computations are performed. The remaining cores are used for all the real-time operations. The software running on these cores is referred to as the Real-Time Controller (RTC).

2.2 Customer Application and Control Loop Components

As indicated previously, a customer creates their own application description. Within the NRTC a software model of all controllable hardware (drives, sensors, and actuators) is available and can be requested through the API. Figure 2.3 presents the software model for the controllable hardware. Within this model, the software representation of a hardware drive is called a *sub-controller*. As mentioned earlier, a drive provides an interface to actuators and sensors within the system. The *sub-controller* software model is mainly used to identify a drive, moreover acts as a container for the software representations of the connected actuators and sensors, which are known as Control Loop Components (CLCs). A customer is able to add control algorithms (e.g. a PID controller implementation) to a specific sub-controller. Prodrive provides the customer a library of generic algorithms, however, a customer may also create their own implementations. These algorithms are also represented as CLCs. All CLCs have something in common, that is, each CLC has either inputs, outputs, or both. After all CLCs are specified, a customer is able to connect the inputs and outputs of each CLC within a sub-controller, moreover across sub-controllers. Typically the amount of connections between CLCs in a sub-controller is greater than the amount of connections between CLCs across sub-controllers.

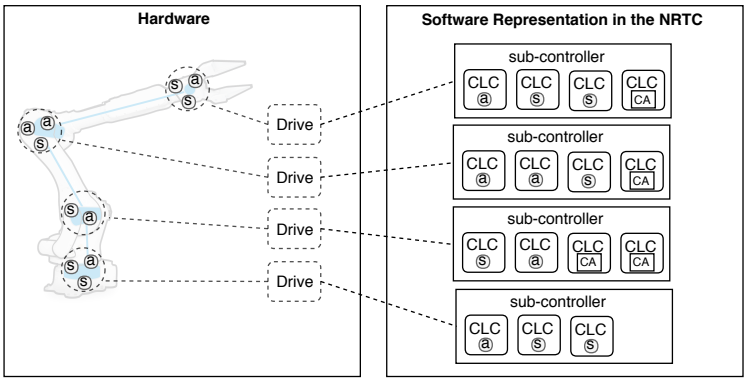


Figure 2.3: Software representation of the controllable hardware.

The CA abbreviation stands for a software implementation of a Control Algorithm.

2.3 From Customer Application to Dependency Graph(s)

Within the NRTC, the customer application, which includes all CLCs and their connections, is transformed into a task-set consisting of elementary computations and operations. In order to preserve application behaviour as intended by the customer, the task within this set should be executed in a specific order. To represent dependencies between tasks, a dependency graph for the task-set is created, consisting of nodes (tasks) and edges (dependencies) as presented in Figure 2.4.

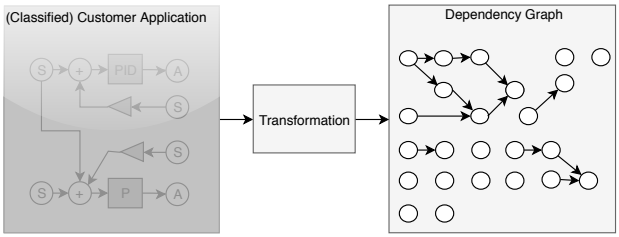


Figure 2.4: Transforming the customer application into a dependency graph.

After transforming the customer's application into a dependency graph, the graph is split into a communication critical and non-communication-critical part as depicted in Figure 2.5.

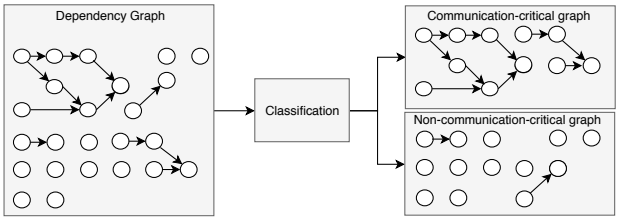


Figure 2.5: Splitting the dependency graph into a communication critical and non-communication-critical part.

The communication-critical task-set consists of tasks that directly act on data retrieved from the hardware drives (e.g. sensor data), after which the processed data is send to the hardware drives (e.g. actuator data) within the same execution cycle. The non-communication-critical task-set consists of tasks that, either act on data received from the drives which is not send to the drives within the same cycle, the other way around, or when neither the data received; nor send to the drives is altered. All possible classifications are presented in Figure 2.6.

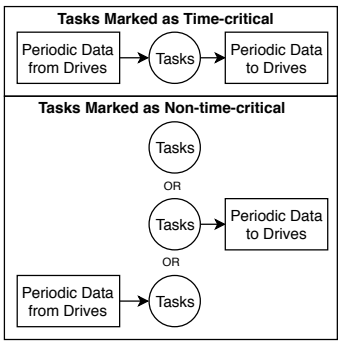


Figure 2.6: Time Criticality Classification of Tasks Within PMP.

2.4 Execution Cycle

The application specified by the customer, is meant to be executed in a periodic fashion. The application description also includes a throughput requirement expressed in terms of the required control frequency, which translates in the maximum period of time every execution cycle of the entire application should be finished.

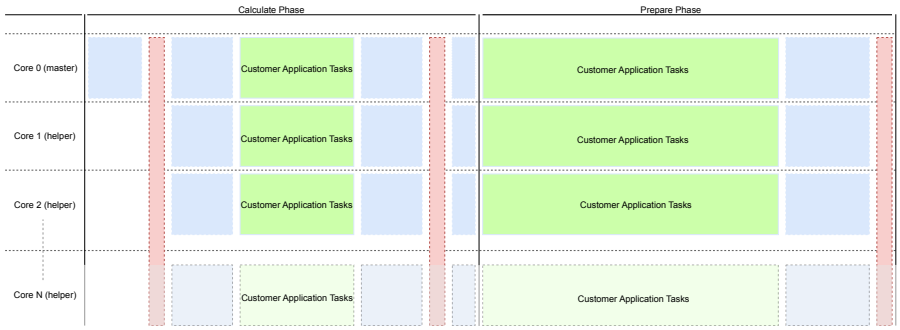


Figure 2.7: Schematic overview of an execution cycle.

Figure 2.7 depicts a schematic example of an application execution cycle within a PMP system. Each execution cycle is divided into two phases, namely, calculate and prepare. Within the calculate phase all communication-critical tasks are handled, in the prepare phase all non-communication-critical tasks are handled. Within Figure 2.7, the green bars relate to the customer's application, whereas the blue bars relate to PMP itself. The schedule and assignation of these PMP intrinsic tasks may not be altered and are therefore left out of scope. The red coloured bars represents necessary (multi-core) synchronization points on task-set level, i.e. the communication to the drives can only be initiated when all tasks within the calculate task-set have been executed. In order to resolve dependencies within the task-sets themselves, a more fine grained *producer* and *consumer* model is used.

2.5 Consumer Producer Model

PMP includes a *consumer* and *producer* model to resolve core-to-core dependencies within a task-graph. Consumer and producers are represented by tasks themselves, and inserted into the task-graph when required (see Figure 2.8).

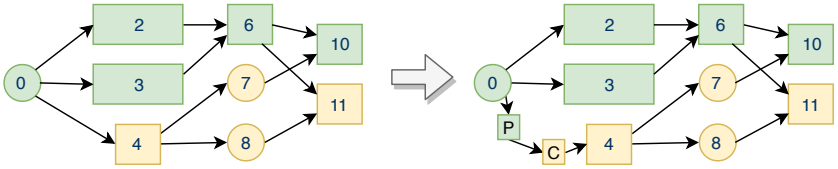


Figure 2.8: Insertion of producers and consumers.

Figure 2.9 depicts an example where dependencies are resolved using producers and consumers, during execution of the task-graph in Figure 2.8. Within the example, green tasks are assigned to *Core 0* and yellow tasks to *Core 1*.

Task 4 on *Core 1* depends on *Task 0* mapped to *Core 0*, because they share a portion of data; therefore a producer task is added after *Task 0*, and a consumer task is added before *Task 4*. The consumer is only allowed to continue when there is a token to consume, which is produced by the producer task after *Task 0*. In this way, *Task 0* is always executed before *Task 4*, resolving the dependency.

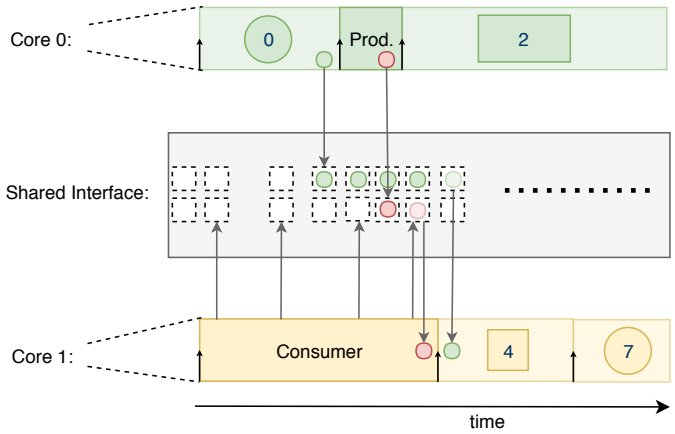


Figure 2.9: The use of Producers and Consumers to resolve dependencies.

The main advantage of the producer and consumer method is its simplicity, i.e. a consumer or producer is just another task. The downside however, is the fact that two additional tasks need to be inserted, which causes execution overhead. In addition, the producer and consumer model requires core-to-core communication, which is costly.

2.6 Deadline and Real-time Specification

Within the complete task-set there exists only one explicitly defined deadline, that is, the desired cycle period of the customer's application. Within this time period, all tasks corresponding to the customer's application, should be executed. All tasks within the complete task-set share this deadline, e.g. there is no explicit deadline specification at task-level.

PMP optimizes for average case execution efficiency and can be categorized as firm real-time. Missing a deadline is not catastrophic, though, frequent -and subsequent misses in particular- can not be tolerated due to customer guarantees and possible system damage. In order to meet the throughput demands, it is necessary to exploit parallelism in the task-sets by means of scheduling and assigning the tasks to the available multi-core hardware resources, without violating any of the dependencies.

2.7 Task Assignation

Task assignation is the process of mapping tasks to hardware resources. Within PMP, this assignation is based on a manual optimization process at sub-controller level. As mentioned before, a sub-controller is the software representation of a hardware drive in the NRTC, which includes sensors and actuators. Within a customer application, sensors and actuators can be arbitrarily connected, moreover, control algorithms can be inserted. All tasks corresponding to sensors, actuators, and control algorithms, belonging to a specific sub-controller, can be assigned to a certain core in the system. Because a customer is also able to connect actuators and sensors across drives, the task-sets belonging to each sub-controller, can also be connected to each other. Figure 2.10 depicts an example assignation of two task-sets (including dependencies) corresponding to two different sub-controllers.

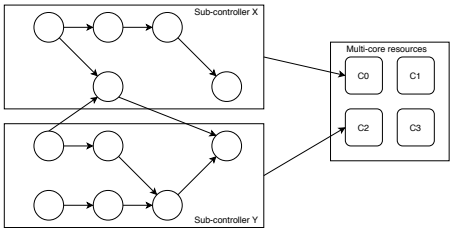


Figure 2.10: Sub-controller task assignation.

The main idea behind this manual process is that, within a PMP application, the task-set belonging to a sub-controller has a high dependency density when compared to the complete task-set, thus providing interesting candidates for (manual) task parallelization. The assignation is described in a product specific XML configuration file. When the configuration is omitted a heuristic takes over, which tries to equally divide all sub-controllers among the available hardware resources, in a round-robin fashion.

> In order to distinguish the two methods, the manually optimized variant will be referred to as *mapped* whereas the heuristic will be referred to as *non-mapped*.

In order to manually optimize the assignation, a customer typically reveals parts of the application, which includes the connections between CLCs. Using this information, Prodrive engineers try to divide the tasks (on sub-controller level) as equal as possible through trial and error procedures, using the resulting cycle execution time as a guide. Besides the fact that this process is time-consuming, it may also result in poor assignation if (parts of) the customer's application is unavailable, or the application is obfuscated too much, in which it does not resemble the "real" application anymore. Moreover, a customer is allowed to change the entire application, which, if failing to meet the throughput requirements, requires another trial-and-error iteration at Prodrive, which is undesired. In addition to these problems, the manual assignation approach provides limited tuning possibilities, e.g. even though the tasks within current products

can be assigned reasonably well at sub-controller level, new products might not be evenly assignable at sub-controller level. Given that the execution time is not measured at task-level, in combination with that fact that a customer is able to implement her/his own control algorithms, the performance as perceived by the customer, may be far from optimal.

2.8 Task Scheduling

After the assignation process, a schedule is determined for each core. In the current implementation, a schedule is created on the NRTC solely based on the existing dependencies, and executed on the RTC during run-time.

Depending on the sub-controller-to-core assignation, each set of connected nodes that is assigned onto a single core, is called an execution path. All execution paths that are connected together form an execution group. Figure 2.11 presents an example of an execution path and an execution group. Nodes with the same colour, represent nodes assigned to the same core. Since depended execution paths in a group can be assigned onto different cores, producer and consumer nodes are inserted into the graph (see Figure 2.12).

For each execution group, the end-nodes of each execution path in the group, are sorted; in which an end-node represents a node without having a successor assigned to the same core, i.e. without having a successor within the same execution path. For the green coloured execution path in Figure 2.12, the end-nodes are 12 and 10. After identifying all end-nodes, the following scheme is used to schedule the nodes in each execution path:

- All end-nodes representing a producer node are scheduled first.
- Then end-nodes consisting of neither a producer or consumer.
- And last the end-nodes representing a consumer.

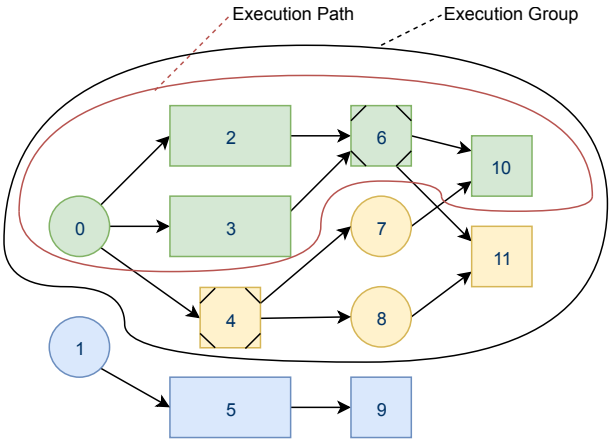


Figure 2.11: Execution path and execution group.

Nodes having the same colour, represents nodes assigned to the same hardware resource.

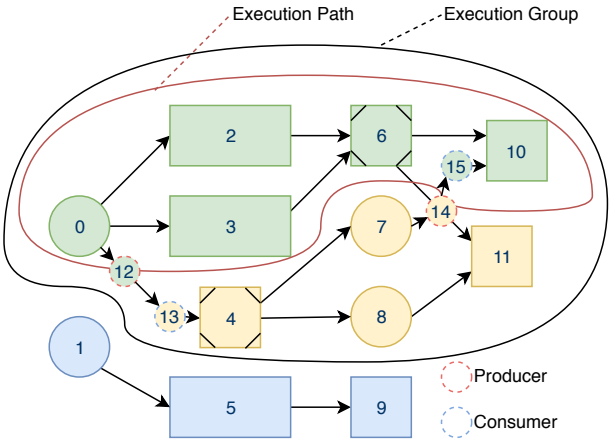


Figure 2.12: Execution path and execution Group including producers and consumers.

Nodes having the same colour, represents nodes assigned to the same hardware resource.

Using this scheme, only dependency information is used to determine a schedule on the NRTC, metrics like computational load are not being accounted for.

After the schedule and assignation are determined for each task, this information is communicated once to the RTC, where the schedule is followed every cycle using a simple online scheduler. The advantage of having a simplistic online scheduler on the RTC, is the minimal processing and memory overhead; the downside however is the inability to change the execution order during run-time.

2.9 Conclusion

In this chapter the original scheduling and assignation procedure was discussed. It was shown that assignation was carried out at sub-controller level, in which a sub-controller represents the software model of a connected hardware drive. Furthermore it was shown that the assignation process is optimized by hand and is therefore time-consuming. Without specifying a manually optimized configuration, it was shown that a round-robin heuristic is followed. Neither of the two methods take into account core-to-core communication costs or the computational intensity of each task. Our analysis clearly indicates that the manual approach will fail to provide a practical and sustainable solution for future PMP products, which are becoming increasingly sophisticated.

In the next chapter, the necessary terminology regarding (multi-core) scheduling is discussed, whereafter an overview of state-of-the-art multi-core scheduling solutions, is presented.

3 Background and Preliminaries

Multi-core scheduling and assignation algorithms have been a topic of research for many years, moreover for many years to come. The increase in performance of single-core architectures in their current form have reached both physical and practical limits. The past twenty years, multi-core architectures have gained a dominant position in the use of computing systems due to their advantages in both speed and power usage. As a consequence, single-core processors for general-purpose computers are not even produced anymore. This trend is also observable in the design of Embedded Systems. More and more embedded solutions are designed using multi-core platforms. Designing of (embedded) systems that can utilize a multi-core platform in an efficient way is a complex problem, especially given real-time deadlines, execution dependencies, unpredictable hardware, etc.

In this chapter, relevant concepts concerning (multi-core) scheduling, including necessary terminology, are introduced. In addition, state of the art scheduling algorithms are presented.

3.1 Scheduling Terminology

As mentioned in Chapter 1, mapping *tasks* onto hardware resources is called *assignation*, moreover determining the execution order is known as *scheduling*. In this section, all concepts concerning multi-core scheduling and assignation will be discussed. Furthermore an overview is given of the terminology and conventions that will be used throughout this thesis.

3.1.1 Tasks

Tasks are the units of computation that requires processing in order for the system to operate correctly. A task is often described having a (possibly varying) computational load (also called task length, or task weight). Tasks are often periodic, meaning the task is to be processed on a regular interval. These tasks often have deadline and period specifications. On the contrary a task can also be sporadic, meaning the task requires processing on irregular intervals. Such tasks often have deadline and start-time specifications. Tasks can also have additional properties such as a priority. If a task can be interrupted during execution by another task, it is said to be preemptive. If a task cannot be interrupted, it is called non-preemptive. A collection of tasks in a system is called a task-set. Because dependencies may exist between tasks (e.g. because of data exchange), a task-set is often modeled as a graph in which the nodes represent tasks, and edges dependencies.

3.1.2 Directed Acyclic Graph

Figure 3.1 shows an example task-set modeled as a graph. Since the graph in Figure 3.1 does not contain any cycles, moreover all edges are unidirectional, it is better known as a Directed Acyclic Graph (DAG).

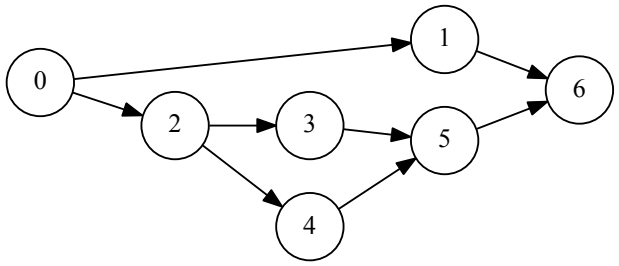


Figure 3.1: A Directed Acyclic Graph of a task-set.

In this thesis a DAG will be defined as a tuple $G = (N, E, W_E, W_N)$ where:

- N is the set of all Nodes which represent tasks, where $n_x \in N$ represents a Node with index x.
- E is the set of all Edges which represent dependencies, where $E \subseteq N \bigtimes N$, in which $e_x = (n_u, n_v) \in E$ represents an Edge with index x.
- $W_E : E \to \mathbb{R}_{\geq 0}$ is the edge weight mapping.
- $W_N : N \to \mathbb{R}_{\geq 0}$ is the node weight mapping.

The DAG defined above models processor-to-processor communication using edge weights, which represent communication cost between two nodes, given they are executed on different processors.

3.1.3 Scheduling, Assignation, and Clustering

A *schedule* specifies the order and resource assignment of each task from the task-set. The process of assigning a hardware resource (e.g. a processor core) to a task is known as *Assignation*. Determining the execution order of all tasks is known as *scheduling*. Throughout this thesis, a *scheduler* is defined as an algorithm that performs both *assignation* and *scheduling* operations.

Depending on the hardware platform and requirements, different assignation and scheduling policies can be used. Assignation and scheduling of a task-set can be performed during different stages of system operation, depending on the desired behaviour. In the remainder of this thesis, the following operating stages will be used:

Compile-time (offline) Schedules that are determined during compilation of the code are called compile-time schedules.

Configuration-time (offline) Schedules that are determined after booting the system, before entering the run-time phase, are called configuration-time schedules. The advantage compared to compile-time is that configurations can be altered without having to recompile the code.

Run-time (online) As the name implies, run-time schedules are determined at run-time. The main advantage of a run-time scheduler is the ability to cope with execution time variations.

If a schedule is fixed during run-time it is called static. Moreover if a schedule can change during run-time it is called dynamic.

The set of available processors for scheduling is indicated using P, with $p_x \in P$ a single processor with x as identification number. Grouping nodes that are assigned to the same core is known as clustering and the groups are called clusters. A cluster can either be non-linear, when two (or more) independent nodes are mapped onto the same cluster, or linear. Equation (3.1) shows the notation convention for all clusters at iteration i. Given a graph g which includes a set of nodes $N = \{n_0, n_1, n_2, n_3, n_4, n_5\}$ and given a set of processors $P = \{p_0, p_1, p_2\}$ then the clusters for this graph $C_i(P)$ at some iteration i could for example look like: $C_i(P) = \{\{n_0, n_3\}, \{n_2\}, \{n_5, n_4\}\}$, where ordering in each cluster, indicates execution order on the corresponding processor. Equation (3.2) is used to get the processor p_x for a clustered node n_x.

$$C_i : P \to \wp(N) \tag{3.1}$$

$$CID_i : N \to P \tag{3.2}$$

As mentioned before, an ordered sequence of nodes is referred to as a schedule. In multi-processor scheduling, a schedule $S_i(Ci(p_x))$ is constructed for each cluster $Ci(p_x)$. A schedule $S_i(Ci(p_x))$ contains the starting times $t_{start}(n_x)$ for each node n_x within a cluster $Ci(p_x)$ on processor p_x at iteration i. Given a schedule $s = S_i(Ci(p_x))$, the schedule length $LEN_i(s)$ at iteration i is defined as the node with the latest starting time including node weight on processor p_x, which is found in Equation (3.3).

$$LEN_i(s) = max(\{t_{start}(n_x) + W_N(n_x) : n_x \in s\}) \tag{3.3}$$

3.1.4 DS, CP, ST and PT

DS *The* longest path in a *scheduled* DAG is better known as the Dominant Sequence (DS).

CP *A* longest path in the clustered (not yet scheduled) DAG is referred to as a Critical Path (CP).

ST The Sequential Time (ST) of a DAG equals the sum of all task weights without including the edge weights, which depicts single processor execution.

PT The Parallel Time (PT) of a DAG is determined by the Dominant Sequence which depicts the total execution weight given the required multi-processor resources.

3.1.5 Successors and Predecessors

For every node in the graph, *successors* and *predecessors* are defined by Equation (3.4) and (3.5) respectively. The successors of a node n_x, defined by $SUCC(n_x)$, consists of a unique set of nodes, for which all nodes in this set, there exist an edge that originates from n_x. Likewise the predecessors of a node n_x, defined by $PRED(n_x)$, consists of a unique set of nodes, for which all nodes in this set, there exist an edge with destination n_x. The origin and destination of an edge e_x is indicated with $src(e_x)$ and $dst(e_x)$ respectively.

$$SUCC(n_x) = \{N' : N' \subseteq N : \forall n_y[n_y \in N' : \exists e_x[e_x \in E : src(e_x) = n_x]]\} \tag{3.4}$$

$$PRED(n_x) = \{N' : N' \subseteq N : \forall n_y[n_y \in N' : \exists e_x[e_x \in E : dst(e_x) = n_x]]\} \tag{3.5}$$

The *Ready List*, $Ready_i$, is defined as the list of all nodes that can be scheduled at some iteration (or time-step) i. A node n_x resides in the *Ready List*, $Ready_i$, if it is not already scheduled and all of its predecessors $PRED(n_x)$ have been scheduled.

3.1.6 Top- and Bottom-levels

Given a Directed Acyclic Graph g, then the *top-level* $tl(n_x)$ of a node n_x in the graph is defined as the length of the longest path from an entry node to n_x *without* the node cost of n_x (see Equation (3.6)). The *bottom-level* $bl(n_x)$ is the inverse and depicts the length of the longest path from n_x to an exit node *including* the node weight $W_N(n_x)$ (see Equation (3.7)). The bottom-level of a bottom-node (i.e. a node without successors) equals its weight and the top-level of a top-node (i.e. a node without predecessors) equals zero. If node and edge weights are static parameters, the bottom- and top-levels for each node do not change in between scheduling steps. In most literature, bottom- and top-levels that do not change are identified as static. The top- and bottom-nodes of a graph g are described using Equation 3.8 and 3.9. Using the bottom-levels of every node, the Critical Path length in the graph can be calculated using Equation (3.10), which seeks the highest occurring bottom-level for all nodes in the graph.

$$tl(n_x) = \begin{cases} 0 & \text{if } n_x \in T(g) \\ max\{tl(n_y) + W_N(n_y) + W_E(n_y, n_x); n_y \in PRED(n_x)\} & otherwise \end{cases} \tag{3.6}$$

$$bl(n_x) = \begin{cases} W_N(n_x) & \text{if } n_x \in B(g) \\ max\{bl(n_y) + W_N(n_x) + W_E(n_x, n_y); n_y \in SUCC(n_x)\} & otherwise \end{cases} \tag{3.7}$$

$$T(g) = \{n_x \in N : PRED(n_x) = \emptyset\} \tag{3.8}$$

$$B(g) = \{n_x \in N : SUCC(n_x) = \emptyset\} \tag{3.9}$$

$$CP(g) = max\{bl(n_x) : n_x \in N\} \tag{3.10}$$

3.1.7 List and Critical Path Scheduling

List scheduling is one of the most commonly used scheduling approaches for Directed Acyclic Graphs. In a list scheduling algorithm, nodes are assigned priorities based on some metric, and then ordered priority wise. During the creation of a schedule, the highest priority unscheduled node is examined and scheduled if possible, after which the next (lower priority) node is examined. The bottom levels for each node is an often used metric within list scheduling. This method is based on the intuitive assumption that nodes with higher bottom-levels, should in general be executed first. This method is known as Critical Path scheduling.

3.2 State of the Art Scheduling Algorithms

In this section several multi-core aware scheduling algorithms are presented, which could potentially be utilized in order to solve the problem stated in Section 1.1.

3.2.1 Communication Unaware Scheduling Algorithms

First some (early) multi-processor (list) scheduling algorithms are investigated which do not take into account any communication cost. Even though communication costs are not taken into account, the multi-processor scheduling problem remains NP-Complete, thus some kind of heuristic is needed in order to generate (sub-optimal) schedules in polynomial time. Scheduling without taking into account communications can be helpful when communication timings are not (explicitly) known or known to be of unit size.

3.2.1.1 The Intuitive Approach

The most intuitive and simplistic approach is to calculate all static bottom-levels for every node in the graph using Equation (3.7), without considering edge weights W_E. Then, given a Bounded Number of Processors (BNP), schedule all nodes in order of decreasing bottom-levels, in round-robin style, to the first idle processor in the set of available processors. The intuitive approach is described using the following steps:

S1 Calculate the (static) bottom-levels for every node using Equation (3.7), omitting the edge weights W_E.

S2 Schedule an unscheduled node with the highest bottom-level to the first processor that is idle, if a tie occurs, solve it at random.

S3 Repeat step **S2** as long as there are unscheduled nodes left, otherwise terminate.

The intuitive approach has a time complexity of $\mathcal{O}(N)$. In the first step the bottom-levels are calculated which takes N steps, afterwards all nodes are scheduled one after the other, which also takes N steps. Bear in mind that this is a rather coarse indication of the total complexity since the ordering of bottom-levels (which is a general sorting problem) already has a worst-case time complexity of at least $\mathcal{O}(N \log N)$, depending on the algorithm that is used.

3.2.1.2 Highest Level First with Estimated Times

The Highest Level First with Estimated Times (HLFET)[5] uses the Earliest Start Time (EST) metric to produce a schedule. HLFET assumes a bounded number of processors (BNP). The algorithm can be described by the following steps:

S1 Calculate the (static) bottom-levels for every node using Equation (3.7).

S2 Schedule an unscheduled node with the highest bottom-level to the processor that allows the Earliest Start Time(EST) using Equation (3.11).

S3 Repeat step **S2** as long as there are unscheduled nodes left, otherwise terminate.

Every iteration, the HLFET algorithm performs an assignation and scheduling operation at the same time, because the order of assigning a node n_x to a processor p_x, determines the final execution order on this particular processor. The clustering notation from 3.1 can be used to represent the schedules for every processor $p_x \in P$ at iteration i.

The Earliest Start Time(EST) for a node n_x scheduled to a processor p_x in iteration i (which is used in step **S2**), is calculated using Equation (3.11), which basically says that the Earliest Start Time is, either the maximum over all ESTs for predecessor nodes $PRED(n_x)$ including the node weight $W_N(n_y)$ of the predecessors, or the current schedule length $LEN(S(C_i(p_x))$ on processor p_x. The EST can be seen as a top-level calculation that takes into account the already scheduled nodes. Equation (3.12) shows the EST equation, if a node is to be scheduled to a particular processor. This equation is used in Equation (3.11), to find the processor allowing the lowest EST.

$$EST_i(n_x) = MIN(\{EST_i(n_x, p_x) : p_x \in P\}) \tag{3.11}$$

$$\begin{gathered} EST_i(n_x, p_x) = max(max_{pred}, LEN(S(C_i(p_x)))) \\ where: \\ max_{pred} = \begin{cases} max(\{EST_i(n_y) + W_N(n_y) : n_y \in PRED(n_x)\}) & if PRED(n_x) \neq \emptyset \\ tl(n_x) & otherwise \end{cases} \end{gathered} \tag{3.12}$$

The worst-case time complexity of HLFET is $\mathcal{O}(N^2)$.

3.2.2 Communication Aware Scheduling Algorithms

Often task dependencies occur between tasks that exchange data. When two tasks share data and are scheduled to the same processor, this data is (immediately) available due to local data caches. However, when two tasks that share data, are scheduled onto different processors, this data has to be transfered in some way to the other processor before the other processor can start executing. This communication is often carried out via a relatively slow shared interface, therefore introducing communication overhead.

The following section gives an overview of multi-processor scheduling algorithms that take into account this communication overhead, by representing these overheads with edge weights W_E. The basic principal that is used in communication aware scheduling algorithms, is that, edge weights are zeroed out when two tasks are scheduled onto the same processor.

3.2.2.1 The Intuitive Approach

Much like the approach described in Section 3.2.1.1, the intuitive approach is to sort all nodes using the bottom-levels and schedule each node to the first idle processor that is available. However the definition of a bottom-level in Equation (3.7) does not consider any edge-zeroing when two tasks are scheduled onto the same processor, so performance ought to be poor.

3.2.2.2 Insertion Scheduling Heuristic

The Insertion Scheduling Heuristic [6] by Kruatrachue is a list-scheduling algorithm that uses an *insertion* approach. This means that in every iteration the algorithm tries to fill the idle time slots (if there exist any) in the schedule. The ISH algorithm uses Equation (3.7) to calculate the static bottom-levels (omitting the edge weights W_E). The algorithm schedules each node in order of decreasing bottom-levels to the processor that allows the EST. In order to take into account communication costs, Equation (3.11) is adapted with edge weights, which forms Equation (3.13). Furthermore the EST equation for a node n_x, that is to be scheduled onto a processor p_x, is redefined using Equation (3.14). The algorithm consists of the following steps:

S1 Calculate the (static) bottom-levels for every node using Equation (3.7), omitting the edge weights W_E.

S2 Construct a *Ready List*, $Ready_i$, by adding all top nodes $T(g)$ in the graph g.

S3 Sort all nodes in the *Ready List*, $Ready_i$, in decreasing order of the bottom-levels calculated in **S1**, if a tie occurs, solve it at random.

S4 Schedule the first node in $Ready_i$ to the processor that allows the Earliest Start Time using Equation (3.13).

S5 If a gap / idle period is introduced on the processor p_x to which the node in **S4** is scheduled, traverse the *Ready List*, $Ready_i$, and schedule as many nodes as

possible within the idle time slot that cannot be scheduled earlier on any other processor.

S6 Update the *Ready List*, $Ready_i$, for the next iteration $i = i + 1$. Goto **S3**.

$$EST_i(n_x) = MIN(\{EST_i(n_x, p_x) : p_x \in P\}) \tag{3.13}$$

$$EST_i(n_x, p_x) = max(max_{pred}, LEN(S(C_i(p_x))))$$
$$where:$$
$$max_{pred} = \begin{cases} max(\{EST_i(n_y) + W_N(n_y) : n_y \in PRED(n_x)\}) & \text{if } PRED(n_x) \neq \emptyset \\ & \text{and } n_y \in C_i(p_x) \\ max(\{EST_i(n_y) + W_N(n_y) + W_E(e(n_y, n_x)) : n_y \in PRED(n_x)\}) & \text{if } PRED(n_x) \neq \emptyset \\ & \text{and } n_y \notin C_i(p_x) \\ tl(n_x) & otherwise \end{cases} \tag{3.14}$$

The ISH algorithm has a time complexity of $\mathcal{O}(N^2)$.

3.2.3 Clustering Algorithms

Algorithms that try to group nodes in so called clusters, are know as clustering algorithms. At the beginning, every node is assigned a unique cluster. Then iteratively, clusters are merged based on some metric or function that is optimized for. During each merge, nodes in the clusters may require reordering depending on the clustering method. Most of these algorithms assume unlimited resources and are categorized as Unbounded Number of Processors (UNP) algorithms. Because of this, each clustering algorithm requires an additional step where clusters are mapped to the available processors. Section 3.2.3.4 provides an overview of algorithms that can be used to assign a clustered graph to a BNP (Bounded Number of Processors).

Clustering algorithms use a slightly extended version of the bottom and top-levels defined earlier. When two nodes n_x and n_y reside in the same cluster ($C(n_x) = C(n_y)$), the weight of the edge between these nodes is set to zero, because of this, the bottom- and top-level values are now "dynamic" since they may change due to clustering. The corresponding equations can be found in Equation (3.15) and (3.16).

$$bl(n_x) = \begin{cases} W_N(n_x) & \text{if } n_x \in B(g) \\ max\{bl(n_y) + W_N(n_x) + W_E(n_x, n_y); n_y \in SUCC(n_x)\} & \text{if } CID(n_x) \neq CID(n_y) \\ & and\ n_x \notin B(g) \\ max\{bl(n_y) + W_N(n_x); n_y \in SUCC(n_x)\} & \text{otherwise} \end{cases} \tag{3.15}$$

$$tl(n_x) = \begin{cases} 0 & \text{if } n_x \in T(g) \\ max\{tl(n_y) + W_N(n_x) + W_E(n_y, n_x); n_y \in PRED(n_x)\} & \text{if } CID(n_x) \neq CID(n_y) \\ & \quad and\ n_x \notin T(g) \\ max\{tl(n_y) + W_N(n_x); n_y \in PRED(n_x)\} & \text{otherwise} \end{cases} \tag{3.16}$$

3.2.3.1 Internalization

Sarkar's Internalization algorithm[7] merges clusters based on the edge weights in the graph. Each iteration the edge with the largest weight is considered and the attached clusters are merged if the Parallel Time (PT) does not increase. After clustering, the nodes residing in the newly formed cluster(s) are order based on the bottom levels before clustering. If necessary, additional zero-weighted edges are added to resemble node ordering in the clusters, such that new bottom and top-levels can be calculated. The (simplified) clustering algorithm is described by the following steps:

S1 Sort all edges in descending order of edge weights.

S2 Calculate bottom and top-levels for each node.

S3 Examine the unexamined edge with the highest edge weight, terminate when there is none.

S4 Merge clusters connected by the edge and order nodes by bottom-levels (adding virtual edges if necessary).

S5 Calculate new bottom and top-levels including PT (which change due to the added virtual edges and clustering) .

S6 If PT not increased, goto step **S3**.
Else, un-merge clusters and restore PT, bottom and top-levels, goto step **S3**.

The time complexity of Internalization as described in [7], is $\mathcal{O}(E(E+V))$.

3.2.3.2 Linear Clustering

The Linear Clustering (LC) algorithm from Kim and Browne [8] tries to decrease the Parallel Time by grouping all nodes belonging to a longest path, into a single cluster. The algorithm is described by the following steps:

S1 Construct an unexamined list, containing all edges.

S2 Select *a* longest path in the graph containing only unexamined edges.

S3 Cluster the nodes in the longest path if the PT does not increase.

S4 Remove all edges in the path from the unexamined list.

S5 When all edges are examined, terminate, else goto step **S2**.

Because the LC algorithm only forms linear clusters, that is, there is never a cluster containing two independent tasks, no reordering is necessary like in the internalization algorithm described in Section 3.2.3.1.

3.2.3.3 Dominant Sequence Clustering

Another clustering method, is to iteratively zero out a particular edge of the current longest path in the graph (better known as the Dominant Sequence). Several ways exist to select the particular edge, moreover multiple variants of the DSC heuristic exists. Gerasoulis and Yang performed a comparative study [9] which also includes the aforementioned clustering algorithms, furthermore elaborates upon their own DSC based clustering heuristic [10].

According to this study, their DSC clustering algorithm outperforms the methods mentioned above, without adding to the computational complexity. DSC introduces two types of clusters: unexamined and examined. Examined clusters represent clusters that can only increase in size. On the other hand, the unexamined clusters represent clusters with a single node, that can only be merged into an examined cluster.

A node n_x is said to be *free* if all predecessors reside in an examined cluster, furthermore a node n_x is said to be *partially free* if at least one predecessor resides in an examined cluster and at least one predecessor reside in an unexamined cluster.

For a *free* node, the priority is defines as the sum of its bottom and top-level, calculated using Equation (3.15). The priority of a *partially free* node is defined as the sum of its bottom and *examined top-level*, where the examined top-level only considers predecessor nodes that reside in an examined cluster, instead of all predecessors (which is the usual way of calculating a top-level).

Just like the other clustering algorithms, each node is assigned a unique cluster at the start of the heuristic, only now these clusters are marked as unexamined. Their DSC algorithm is described by the following steps:

S1 Mark all clusters, containing a top node $n_x \in T(g)$, examined.

S2 Determine the *free* node n_x with the highest priority $P(n_x)$ and the *partially free* node n_y with the highest priority $P(n_y)$.

S3 If $P(n_x) \geq P(n_y)$:

(a) Loop over all the predecessors $n_p \in PRED(n_x)$ of n_x.

(b) If adding n_x as the last node in the (examined) cluster $CID(n_p)$ where n_p resides, improves the top-level of n_x, then insert n_x into $CID(n_p)$, otherwise mark the cluster $CID(n_x)$ where n_x resides as examined.

S4 If $P(n_x) < P(n_y)$:

(a) Loop over all the predecessors $n_p \in PRED(n_y)$ of n_y for which it holds that n_p resides inside an examined cluster.

(b) If adding n_y as the last node in the (examined) cluster $CID(n_p)$ where n_p resides, improves the examined top-level of n_y, then "lock" this cluster until n_x becomes free. Now perform the same steps as in **S3** with the added constraint that the cluster $CID(n_p)$ where n_p resides, is not a candidate anymore.

Yang and Gerasoulis reported a time complexity for the algorithm of $\mathcal{O}((N+E)logN)$.

3.2.3.4 Unbounded Number of Clusters to Bounded Number of Processors

So far all the clustering methods described above assume an unlimited amount of processing resources, which is not the case in a practical system. Therefore each of these methods requires an additional step where clusters are mapped onto physical processors. Note that some of these methods actually belong to a clustering algorithm described above, however they can be used interchangeably, thus making it relevant to consider them independently.

The Internalization algorithm described in [7], also describes a way of assigning clusters to physical processors. This method is based on a list scheduler where all nodes N, are ordered using the bottom-levels. The algorithm can be described using the following steps:

S1 Order all nodes by the bottom-levels and put it into a list L.

S2 Decide which processor p_x allows the lowest PT when scheduling $n_x \in L$, if a tie occurs, use the lowest EST.

S3 Schedule the node n_x, including all other nodes N_c belonging to the same cluster, to p_x.

S4 Remove N_c including n_x from L. If $L = \emptyset$ terminate else goto **S2**.

In [11] Wu and Gajski described a processor assignment method that tries to minimize the total amount of communication, given an arbitrary connected multi-processor model. The algorithm is based on *traffic scheduling*, which is described in [12]. In [13] Liou and Palis show the benefits of a two-step clustering and processor assignment method. In their study three different UNC to BNP approaches are presented: Communication Traffic Minimizing (CTM), Load Balancing (LB) and a randomized algorithm. According to their study, clustering in combination with Load Balancing is a simple but effective method for scheduling task graphs onto multi-core architectures.

3.2.4 Duplication Algorithms

Duplication based algorithms deliberately allocate tasks to multiple processors in oder to reduce communication overhead. Since PMP does not allow any duplication of tasks, these algorithms are left out of scope. Curious readers are referred to the following comparison study [14] performed by Kwok and Ahmad, in which six different duplication based algorithms are compared.

3.2.5 Machine Learning Based Algorithms

Machine Learning based algorithms are algorithms that have the ability to learn from experience which computer scientist T. Mitchell formally describes as "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E."[15], which happens to be a quite popular definition in the machine learning field of study. Machine learning can be used to tackle complex (search) problems like the multi-core scheduling problem. In this section, some scheduling algorithms that use different machine learning strategies, are discussed.

Genetic algorithms(GA) are a type of Machine Learning algorithms that are often used in optimization problems that involve large search spaces. GA's are inspired by natural selection and uses concepts like *cross-overs* and *mutations* to find a (local optimal) solution. The solution candidates are called *individuals* and a set of individuals is known as a *population*. Each individual contains a set of properties which is stored inside of *chromosomes*. In order to distinguish each solution, a *fitness function* is defined (for the metric that is optimized for), that evaluates the performance of an individual.

Every iteration (or generation) a new population is formed by recombining and (randomly) mutating the chromosomes of a sub-set of individuals from the previous population. These individuals are picked at random in most of the algorithms, though individuals with a higher fitness have a higher change to be chosen, therefore steering the solution to a (local) optimum.

3.2.5.1 Genetic Scheduling and Allocation

The Genetic Scheduling and Allocation [16] algorithm proposed by Ali et al. is an example of a genetic based algorithm. In GSA each chromosome consists of ordered triplets containing a node identification number, time step, and processor assignment, which are referred to as the *genes*. The values assigned to these genes are called *allels*. The *Time Steps* form the schedule and indicate the scheduled time for each node. Table 3.1 presents an example chromosome.

Table 3.1: An example Chromosome.

node:	n_2	n_3	n_1	n_4
time-step:	2	4	1	3
PA:	p_1	p_3	$p1$	p_2

The algorithm starts by generating an initial population. To increase the chances of success, GSA uses a diverse initial population generated using four basic scheduling schemes. The algorithm uses a two-parent, ordered crossover mechanism where two individuals (called parents) produce a new individual (called the offspring). The cross-over mechanism selects a random splitting point in the chromosome of the first parent A, where the left part is copied directly to the offspring. The other parent B is scanned from left to right, and all missing nodes are added in this order. Whilst adding the missing nodes from parent B, a violation check is performed by observing the corresponding

time-steps of each node. If a violation would occur, the time-step from parent A is taken instead. For a visual example the reader is referred to Figure 4 in [16]

During construction of the offspring's chromosome, the processor assignment is taken from the same parent that adds the corresponding node. Because this could lead to concurrent assignment of the same processor, violating processor assignments are reassigned.

The algorithm uses several types of mutations. The most relevant are, the time-step value mutation and the processor reassignment mutation. In both cases the mutation is only carried out if it does not introduce any violations.

Since GSA is meant for high-level hardware synthesis, the fitness function is defined to minimize execution time, whilst minimizing hardware resources like ALU's, registers, buses. Therefore, this algorithm is not directly applicable in the multi-core scheduling problem presented in this thesis. However, the fitness function can be altered to optimize for other metrics, like reducing the communication overhead, without changing the main concept of the algorithm.

3.2.6 Cache Aware Algorithms

There exist an abundance of scheduling and assignation methods, though, almost none of them considers the effect of cache on the total schedule length, in a direct way. Which is probably due to the fact that modeling cache behaviour is either complex or impossible due to unpredictability. Since most (multi-processor) scheduling algorithms consider the Worst Case task timings (which is necessary for safety critical hard real-time systems), the safest approach is to always assume a cache miss. However, this might drastically reduce the average case performance. The following section gives an overview of some of the few cache-aware algorithms that exist.

3.2.6.1 Cache-Conscious List Scheduling

Cache-Conscious List Scheduling[17] is a cache aware offline multi-processor scheduling algorithm, recently proposed by Nguyen et al. The algorithm takes (private) caches into account by using variable node weights instead of single valued ones. Each node has a collection of weights, where each of these weights depends on the node that has been scheduled previously on the same processor (i.e. the node weight depends on the context). The algorithm focuses both on instruction and data cache re-usage from a high-level point of view, that is, the concept of cache is taken into account without focusing on a specific hardware implementation.

Since CLS uses a set of weights for each node, the weight mapping function W_N is redefined in Equation (3.17). The mapping function takes an additional argument, that is, the node n_y that precedes the current node n_x.

$$W_N : N \bigtimes N \rightarrow \mathbb{R}_{\geq 0} \tag{3.17}$$

Table 3.2 presents an example of node weights with different preceding nodes, where diagonal values represents weights under the condition of no cache re-usage.

Note that the weights in the table presumes that these tasks are executed on the same core. Any execution time advantage found in the table would therefore be related

Table 3.2: Weights for n_x given different preceding nodes n_y.

n_y \ n_x	n_1	n_2	n_3	n_4	n_5
n_1	10				
n_2	15	20			
n_3	20	15	25	25	
n_4	15	20	20	20	
n_5	15	10	10	10	15

This table is a modified version from Figure 3 in [17].

to either, instruction cache re-usage between two tasks sharing some functionality, or data cache re-usage due to shared information between a pair of tasks. The latter would imply that there is an edge in between the tasks, also having a weight.

The algorithm uses a list scheduling heuristic similar to HLFET, which is described in Section 3.2.1.2. The only difference is that CLS uses the Earliest Finishing Time (EFT) instead of the EST, to assign nodes to cores. The EFT is defined in Equation (3.18) and (3.19), where $LAST(p_x)$ denotes the last scheduled node on processor p_x and $W_N(n_x, LAST(p_x))$ denotes the weight of n_x, if scheduled onto processor p_x. Moreover max_{pred} seeks the EFT for predecessors of n_X, which may be scheduled to another processor. The max_{clus} seeks the previous EFT of the processor, which is the EFT of the last scheduled node in the cluster.

$$EFT_i(n_x) = MIN(\{EFT_i(n_x, p_x) : p_x \in P\}) \tag{3.18}$$

$$\begin{gathered} EFT_i(n_x, p_x) = max(max_{pred}, max_{clus}) \\ where: \\ max_{pred} = \\ \begin{cases} max(\{EFT_i(n_y) : n_y \in PRED(n_x)\}) + W_N(n_x, LAST(p_x)) & if PRED(n_x) \neq \emptyset \\ tl(n_x) + W_N(n_x, n_x) & otherwise \end{cases} \\ max_{clus} = \\ \begin{cases} max(\{EFT_i(n_y) \in C_i(p_x)\}) + W_N(n_x, LAST(p_x)) & if C_i(p_x) \neq \emptyset \\ tl(n_x) + W_N(n_x, n_x) & otherwise \end{cases} \end{gathered} \tag{3.19}$$

Figure 3 in [17] portrays a visual example for a small DAG. Although the algorithm as described in [17] focuses on Worst Case Timings, it can be applied to the Average Case as well.

3.2.6.2 Other Cache Aware Algorithms

In [18] a cache-aware assignation algorithm is proposed that considers the Working Set Size (WSS) of tasks. The algorithm tries to evenly distribute the WSSs (or memory

footprint) over the available hardware resources, furthermore tries to assign tasks that share the same WSS, to the same hardware resource in order to reduce cache misses. This algorithm is suited for tasks on thread level that communicated with each-other, without having explicit dependencies (as in a DAG), therefore this algorithm is not suited in this research. In [19] another cache-aware scheduling algorithm is proposed which focuses on schedulability for Worst Case Execution. The algorithm uses Cache Space Isolation to ensure that the WSS of a task is guaranteed to fit into the cache, in order to guarantee the Worst Case Execution time. Since this algorithm specifically focuses on Worst Case schedulability, it is not suitable within this research.

3.2.7 Online Algorithms

Unlike offline schedulers, online schedulers have the ability to deal with variations in execution timings. One of the major disadvantages of an online scheduler is the additional overhead required. This is one of the main reasons why there is not a lot of literature available that concerns with online multi-processor scheduling for (hard) real-time embedded systems. In most cases the additional overhead outweighs the gains of online timing knowledge.

Most of the algorithms mentioned above can be adapted for use in an online scheduler, however most of them introduce significant overheads, which makes them unsuitable in online usage. This section provides an overview of some (low-overhead) schedulers that are specifically designed for online usage.

3.2.7.1 The Intuitive Approach

The most intuitive and simplistic approach is to create a *Ready List*, $Ready_i$, of nodes that are schedulable (i.e. nodes for which it holds that all predecessors are scheduled) at each iteration of the scheduler. Each time a processor becomes "Idle", because it has finished execution of some node n_x, all of its direct successors $SUCC(n_x)$ for which it holds that all predecessors $PRED(n_x)$ are already scheduled, can be added to the *Ready List*. Furthermore a node from the *Ready List* should be immediately assigned to the "Idle" processor and removed from the *Ready List*. In order to take into account a certain metric (for example communication overhead), the nodes in the *Ready List* can be assigned some priority based on this metric in exchange for more scheduler overhead.

3.2.7.2 Dynamic Task Graph Scheduling

In [20] Choudhury et al. describe an online approach which is similar to the intuitive approach. They use a metric called Static Urgency to priorities tasks, which is similar to the definition of a static bottom-level, which is calculated using Equation (3.7). The major difference with the intuitive approach is the fact that their algorithm has an additional routine where communication edges are scheduled onto channels. The channels model inter-processor communication with a bound on the number of available channels in order to simulate contentions.

3.2.8 Statistical Algorithms

Another way to deal with variable task execution times without resorting to an online scheduler, is the use of a statistically based algorithm. Unfortunately "task execution times are often not normally or even continuously distributed and are not easily amenable to analytical analysis"[21] which, according to the analysis described in Chapter 4, also applies to PMP.

In [21] Satish et al. describe a statistically based algorithm that tries to capture both variations due to cache behaviour and different execution traces within tasks. The algorithm stores timing values in a probability distribution table. Each entry in the table consist of an execution time range, including the probability that a particular task will have an execution time within this range. Moreover, the joint probability distribution of task execution times that are dependent, are stored in separate tables.

Using all these (measured) timing values, the scheduler tries to optimize the *makespan* of the schedule. In the multi-processor case, the *makespan* is the same as the Parallel Time (PT). Due to the statistical timing values, the makespan is no longer a single valued metric, but a statistical distribution. The scheduler tries to find the smallest makespan that guarantees that at least $\eta\%$ of the executions will not exceed it. The value η is better known as a percentile. Depending on the desired "guarantees", a larger percentile can be chosen. Note that actual guarantees (as in Worst Case Execution Time (WCET) scheduling) strongly depend on the accuracy of the task timing measurements. The algorithm proposes several methods to tackle the search problem, a list scheduling based heuristic and a simulated annealing approach. However most of the heuristics described in this chapter are applicable. The method described in [21] specifically mentions applicability for heterogeneous systems, however, it is also applicable for homogeneous architectures, which are used in PMP.

3.3 Conclusion

In this chapter, necessary scheduling and assignation terminology was introduced, moreover an overview was given of state of the art (multi-core) scheduling algorithms. In the following chapter, the original PMP scheduling and assignation algorithm is analyzed using two different multi-core PMP products.

Analysis of the Original Solution

4

In order to understand the limitations of the original solution, the behaviour of the system was analyzed. PMP is a generic platform that can be used in countless products and configurations. Since it is impossible to analyze every configuration, it had been decided to do the analysis on typical, already existing PMP products. The results can be used to get a feel of typical execution behaviour in PMP products. This information will eventually be used to form a basis for a possible design.

The idea is to extract dependency and timing information from the system, in order to portray execution behaviour. Since the platform did not contain any timing mechanism for individual tasks, the implementation was extended.

For every product the amount and type of connected hardware drives, is known, however as indicated in Chapter 2, the customer application is typically unknown. The customer decides for example how all Control Loop Components (CLCs) are connected, moreover, decides upon the applied control algorithms. Given that numerous use-cases are possible, it is impossible to analyze them all.

In order to still retrieve valuable information, the software and control engineers at Prodrive have created applications for each product, which are based on typical customer use-cases. These applications are used within this analysis.

For each test-run, data from at least 10000 cycles are gathered in order to retrieve reliable timing information. Afterwards the average, minimum and maximum execution time for every task is calculated. Last but not least the Worst Case Cycle (WCC), i.e. the cycle resulting in the highest total execution time, is analyzed.

To show the impact of predefined sub-controller to core mappings, the performance in terms of total average execution time is analyzed for both the *mapped* scheduler, as well as the *non-mapped* scheduler. In order to determined the execution time, the Parallel Time is extracted from the system, that is, the highest occurring execution time among all available cores in the system.

4.1 PPCx3

PPCx3 is one of the many products which is implemented using the Prodrive Motion Platform and one of the few products that has a multi-core architecture. PPCx3 is an example of a typical industrial-sized mechatronic system: a wafer scanner. Within PPCx3, three PowerPC e500 cores are available for handling tasks (RTC). The cores are clocked at $1.5GHz$ and the cycle frequency is set to $8kHz$. Within PPCx3, there is one available system timer to time task, which is clocked at $\frac{1}{64}$ of the clock frequency. Since a delta measurement is performed by subtracting the measured start time from the measured end time, the maximum achievable precision equals twice the period of

the timer ($\sim 85.3ns$). A summary of all properties is found in Table 4.1. In this section several characteristics of this product are presented.

Table 4.1: PPCx3 and XEONx3 system properties.

Products:	PPCx3	XEONx3
Platform:	PowerPC e500	Xeon D-1500
RTC cores:	3	3
Cycle Frequency:	8 kHz	10 kHz
Timer Precision:	85.3 ns	25.0 ps

4.1.1 Mapped

The first analysis is performed on the PPCx3 system using an XML configuration that is also used in the real system. This XML configuration has been manually tuned such that the system meets all timing requirements.

Figure 4.1a depicts the calculate DAG after the assignation phase. This figure shows that the predefined sub-controller to core assignation is manually optimized in such a way that core-to-core dependencies do not occur.

In Appendix A.1 a table can be found that presents the execution order of all calculate tasks, which in turn portrays the offline assigned schedule that has been constructed by the system. Observing the order it is shown that the current schedule heuristic tries to place sensors up front and actuators at the back, which make sense for typical controller topologies, where the control algorithms depend on sensor input, and actuators depend on the output of a control algorithm. Note that the timing results differ only a single order of magnitude compared to the precision of the system timer that was used (see Section 4.1). Since the precision ought to be enough for the analysis, is was decided not to change the timer implementation during this phase. However to get the best results during scheduling, it may be necessary to change the implementation in order to (temporarily) increase the precision.

Figures 4.1b and 4.1c include the timing diagram for the worst case and average case cycle, using the *mapped* scheduler. The figure shows that the *mapped* variant works well for the average case, because the workload is spread evenly. As indicated in Section 1.1, the automated solution should perform equally well.

4.1.2 Non-mapped

The second analysis is performed on the PPCx3 system, without the use of a manual XML configuration. Figure 4.2a depicts the DAG after the assignation phase, for the non-mapped variant. Comparing Figure 4.2a with Figure 4.1a, one can observe that the non-mapped variant introduces (undesired) core-to-core dependencies. Note that the placement of nodes may differ due to the automated generation process, however the underlying structure is identical.

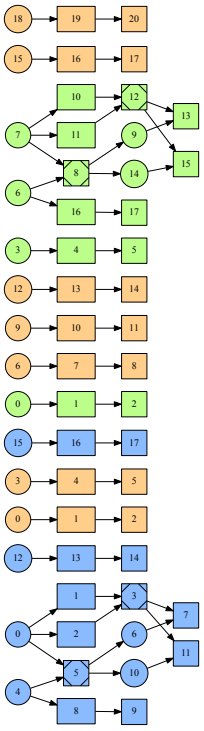
In Appendix A.2 a table can be found that presents the execution order of all calculate tasks for the non-mapped case. As mentioned in Chapter 2, in the non-mapped case,

the assignation algorithm tries to equally divide all sub-controllers over the available cores. However the algorithm does not take into account the amount of tasks (or CLCs) per sub-controller. Therefore the amount of tasks per core, may not be evenly divided. Furthermore, as already indicated in Chapter 2, the computational load of a task is not being accounted for. Last but not least the overhead of the dependency resolving mechanism (better known as the producer and consumer model), is not taken into account.

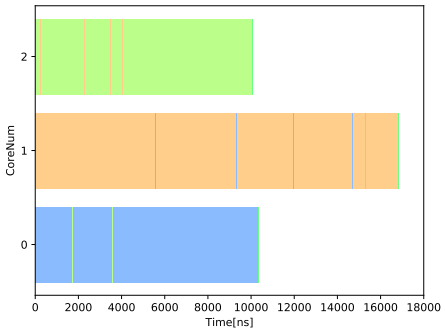
Figures 4.2b and 4.2c include the timing diagram for the worst case and average case cycle using the non-mapped scheduler. Because core-to-core dependencies exist now, additional producer and consumer tasks are present in the timing diagram. It is shown that, due to poor scheduling, *Core 1* immediately starts off with a consumer and therefore has to wait till the producer on *Core 0* produces a token. Furthermore it is shown that when *Core 0* has produced the token, it takes some time for the consumer to continue. This is due to core-to-core communication penalties, i.e. the producer has to load and increment a value stored in the shared memory interface and the consumer (on the other core) has to retrieve this value. By comparing Figure 4.1c with Figure 4.2c it is clear that, on average, the mapped variant does a better job in spreading the workload over the cores.

4.1.3 Prepare Tasks

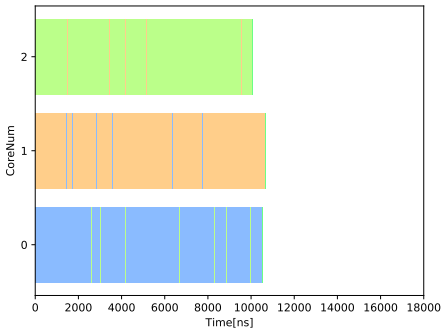
Note that in the previous sections, only the calculate phase of tasks is described. However, as indicated in Section 2.4, there is also a prepare phase. The prepare phase contains a lot more tasks than the calculate phase, however, there are not a lot of dependencies in between tasks. Within PPCx3, there exists 310 prepare tasks in which there exist only two dependencies. Since the visual representation just shows an extensive list of nodes, it has not been included in this analysis. However it does show that both phases may require different scheduling approaches in order to get optimal results.



(a) DAG assigned to the available cores[1].



(b) WCC timing diagram[2].

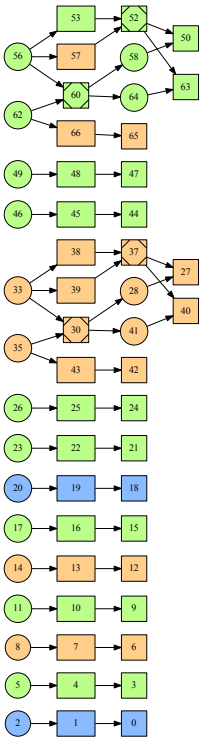


(c) Average timing diagram[2].

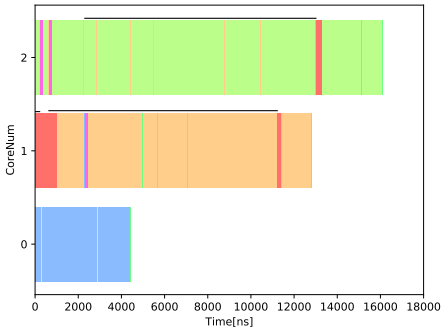
Figure 4.1: Assigned calculate DAG and corresponding timing diagrams for the *mapped* scheduler in PPCx3.

[1]Circles represent *Sensor Tasks*, squares: *Actuator Tasks*, rectangles: *Control Algorithm Related Tasks* and squares with rectangular edges: *Filtering Tasks*. The numbers within nodes represent the execution order on a particular core. Nodes having the same colour are assigned to the same core.

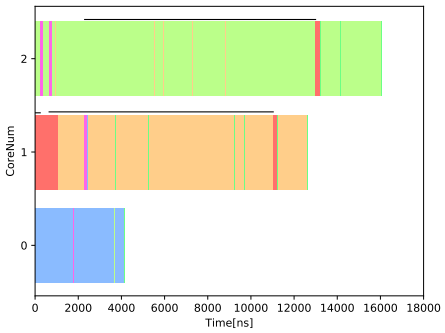
[2]Tasks are represented by bars coloured in the same fashion as the assigned graph shown to the left.



(a) DAG assigned to the available cores[1].



(b) WCC timing diagram[2].



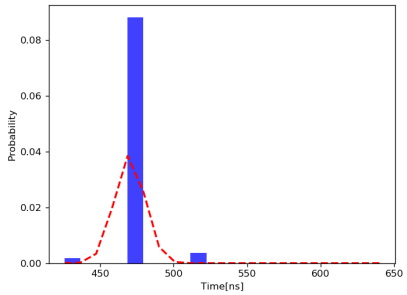
(c) Average timing diagram[2].

Figure 4.2: Assigned calculate DAG and corresponding timing diagrams for the *non-mapped* scheduler in PPCx3.

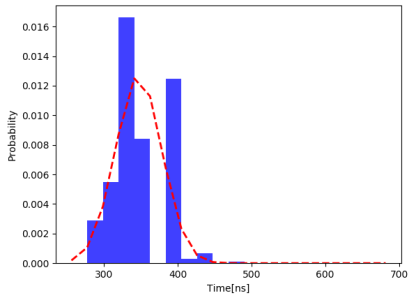
[1]See Figure 4.1 for an explanation of the attributes. Note that consumers and producers -which are tasks themselves- are not explicitly drawn. Thus it seems as if some ordering numbers are missing, whilst in fact these ordering numbers are assigned to the consumer and producer tasks.
[2]See Figure 4.1 for an explanation of the attributes. The lines show the time between the start of a producer and start of the corresponding consumer.

4.1.4 Timing Variation

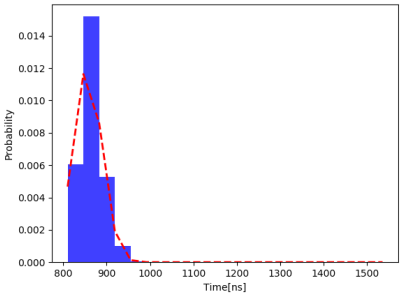
Within PPCx3, there is a variation in task execution times. In order to visualize this, normalized histograms showing the probability of a particular task timing were made for all sub-controller tasks. Figure 4.3 presents some of these histograms that were made. Note that the histograms only show the most significant timing variations in which the variation is higher than the timer precision.



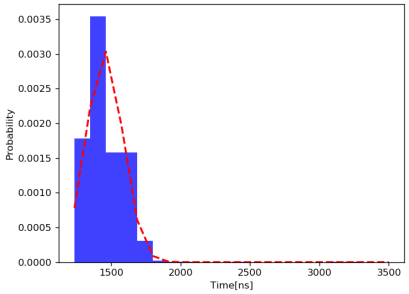
(a) Normalized execution time histogram from one of the actuator tasks in PPCx3.



(b) Normalized execution time histogram from one of the sensor tasks in PPCx3.



(c) Normalized execution time histogram from one of the control tasks in PPCx3.



(d) Normalized execution time histogram from one of the control tasks in PPCx3.

Figure 4.3: Normalized histograms showing the distribution of execution times.

The histograms show that the execution time variations within the system do not follow a typical distribution. There is however a pattern that can be recognized, when comparing different tasks of the same type (like the control algorithm tasks in Figure 4.3) a similar pattern in execution time variation can be observed. After analyzing the code, it was found that some tasks share the same implementation, thus showing similar timing behaviour. For the variation in execution time, the following hypothetical causes were found:

- Conditional statements inside tasks, which depend on the given commands and configuration which can:

  - potentially stall a pipeline due to branch miss predictions.
  - cause additional execution time due to different functional behaviour.

- Data Cache misses

4.2 XEONx3

XEONx3 is one of the latest multi-core products. Just like PPCx3, XEONx3 is an example of a typical industrial-sized mechatronic system: a wafer inspection machine. Since XEONx3 is still in an early development phase, the available hardware is not fully decided upon yet. Development is currently carried out using an Intel Xeon D-1500, which has three available RTC cores clocked at $4.0GHz$. The XEONx3 platform was made available at a late stage during this research. Since this system is interesting nonetheless, it has been added as an additional testing platform. Unlike PPCx3, XEONx3 has no predefined mapping yet. Within XEONx3, there is one available system timer to time task, which is clocked at the core frequency. This means that the timer has a precision of $25.0ps$, which is more precise than the PPCx3 target. In Table 4.1 a summary of all properties can be found. In this section several characteristics will be presented of this system.

4.2.1 Non-mapped

Unlike PPCx3, XEONx3 is more orientated towards a model-based design, which means that a customer has more freedom in altering control algorithm related implementations; thus making XEONx3 more unpredictable compared to PPCx3 in terms of complexity, amount of tasks and graph structure. Within this analysis, a typical application is used in combination with the original schedulers. Since XEONx3 does not have any physical drives yet, both the *mapped* as well as the *non-mapped* schedulers, schedule all task to the first core within the system. Due to the relatively large size, the (assigned) graph of XEONx3 can be found in Appendix A.3. Since XEONx3 currently only uses one core, there is no difference between the original graph and the assigned one. Unlike PPCx3, the graph of XEONx3 is much larger with significantly more dependencies. Furthermore all merge / join nodes (i.e. nodes with more than one incoming edge) carry a lot more edges, which makes it more difficult to parallelize; because all preceding nodes (which may be executed on different cores) need to be finished before the merge node can be executed. Another interesting observation is that the XEONx3 graph contains far less disjoint subgraphs than PPCx3, which again makes XEONx3 more difficult to parallelize in contrast to PPCx3. Last but not least, the graph shows some "redundant" dependencies. Take for example the edge between node 30 and 93. Even though this dependencies does exist (node 93 depends on the output of node 30), the dependency is unnecessary since there exists other paths from node 30 to 93 that are longer.

As mentioned above, XEONx3 has no physical drives yet and thus only a single core is utilized within the *non-mapped* assignation heuristic. Because of this, the average execution timing diagram for XEONx3 has not been included. The single core schedule resulted in an execution time of $7180.24ns$.

4.3 Conclusion

In this chapter two different multi-core PMP products were analyzed. The performance of the original scheduling solutions were presented, as well as the short-comings with respects to new products like XEONx3. In the next chapter the design of a new, automated solution is presented, which, as indicated in Chapter 1, should perform at least equally well compared to the original *Mapped* solution, for both multi-core enabled products.

5 Automated Framework

The achievable performance of the scheduler depends on the accuracy and relevancy of the timing information extracted from the system. However the achievable performance also strongly depends on the way a scheduler makes use of this timing information. Having lots of relevant and accurate timing information in combination with a scheduler that is not able to use it effectively, is just as bad as having an intelligent scheduler that is fed inaccurate information.

Given this insight and the fact that the motion platform is subjected to an ever ongoing development process, it was decided to follow an iterative design and implementation flow. As a first step in the iterative design flow, the scheduling and measuring processes have been decoupled by designing and implementing separate scheduling and measuring frameworks. For both frameworks, an extensive design exploration was performed.

The iterative approach allows for intermediate testing and evaluation. Based on the intermediate results, it was decided if time should be spend in optimizing the measuring framework, or the scheduler framework. In this chapter the design for both frameworks is presented, moreover an overview of all implemented designs is given.

5.1 Framework Overview

In Chapter 2 an overview of the current solution was given. It was shown that there exists two DAGs which require scheduling and assignation, one consisting of communication-critical *calculate* tasks and the other consisting of non-communication-critical *prepare* tasks. Currently the assignation and scheduling process for both DAGs, is based on a manual process. The idea is to design a generic framework which replaces the original scheduling and assignation solution. Thus a framework has to be designed, which, given a DAG and some abstract model of the hardware, produces a valid multi-core schedule, in an automated way. Figure 5.1 depicts an abstract visual representation of the process.

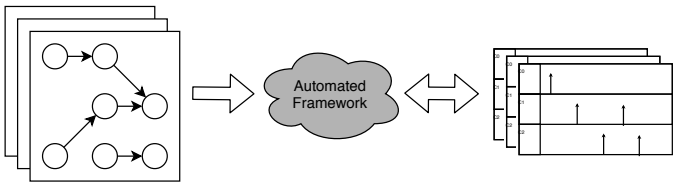


Figure 5.1: From DAG to schedule.

Figure 5.2 presents the design for the automated framework. The red arrow shows the main flow from DAGs to schedules. As indicated in Chapter 2, both calculate and prepare task DAGs are represented using execution paths and execution groups. Since this

representation does not fit within a generic scheduling approach, moreover, unnecessarily complicates the scheduling procedure, it was decided to improve the implementation by using predecessor and successor list in conjunction with nodes and edges.

The first step within the automated flow is the measuring framework. Within the measuring framework, the unweighted DAGs should be transformed into weighted DAGs. The weighted DAGs can then be scheduled using any scheduler within the scheduling framework. Due to the chosen measuring technique (which is explained in the upcoming sections), the online scheduler on the RTC required a partial redesign.

To support the iterative design, moreover make the approach generic, each scheduling implementation is made available in a library, more specific the Extendable Scheduler Library (ESL). The ESL allows to schedule the prepare and calculate DAGs using different scheduling algorithms, moreover allows to easily compare each solution. In order to keep the original scheduling and assignation solution intact, the *Mapped* and *Non-mapped* solutions have also been made available through the ESL.

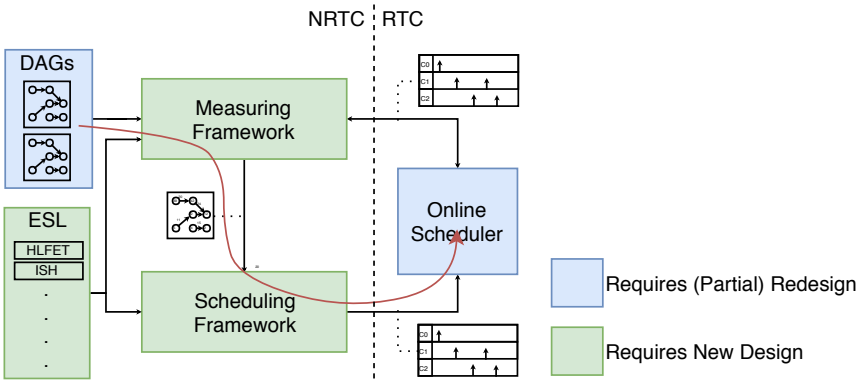


Figure 5.2: Automated framework design.

5.2 Performance Measurement Framework

In this section the design of the performance measurement framework is described. The framework should provide accurate and relevant timing information about all tasks and dependencies within the DAG that has to be processed. An ideal framework would be able to capture all causes of timing variations, such that the exact execution time can be predicted for a certain schedule. However measuring every cause of timing variation is impossible, moreover requires an intelligent scheduler in order to use all the timing information effectively. Thus a selection of the most significant metrics has to be made which influence the total execution time of a particular schedule.

As explained in Chapter 2, the computing hardware, responsible for handling the DAGs of tasks (which constitute the customer's application), typically varies per product. Because of this, an abstract model of the hardware is required, which is depicted in Figure 5.3.

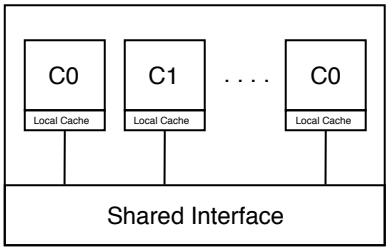


Figure 5.3: Abstract model of the hardware resources.

In the figure the hardware inside the master controller (see Chapter 2), is modelled as a multi-core architecture consisting of a certain amount of cores. Moreover, each core contains a local data and instruction cache, in which the size, caching policy, and amount of layers, is product specific. To allow for core-to-core communication, a shared interface is available within every product. This shared interface may consist of shared caches, shared memory, or any other core-to-core communication enabling concept.

5.2.1 Measurable Metrics

Based on the observations from Chapter 4 and the scheduling literature presented in Chapter 3, a selection of the most significant causes which influence the total execution time, is presented together with the corresponding metric that has to be measured. The abstract model of Figure 5.3 is used in combination with two cores, in order to create all graphical examples.

M1. Intrinsic Task Weight: The first and most obvious cause is the computational demand of each task, which will be referred to as the intrinsic task weight (see Figure 5.4). Within PMP, tasks may differ significantly with respects to computational demand (e.g. a control algorithm task requires in general more computations than a sensor task), thus the intrinsic task weight differs per task. Within a PMP task, conditional statements can be found, which makes the intrinsic weight, a multi-valued metric.

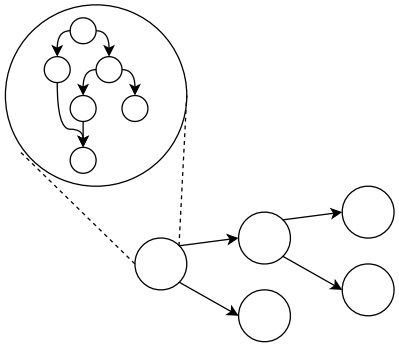


Figure 5.4: Intrinsic task weight.

M2. Core-to-core Overhead for Shared Data: The second cause of execution time variations are due to core-to-core communications, for data shared by dependent tasks assigned to different cores. Figure 5.5 depicts a situation in which the assignation leads to a guaranteed cache miss for the shared data, within the local cache of C_1. If the yellow task in Figure 5.5, was scheduled on C_0, a cache hit would be expected. In this situation, it is in theory still possible that a cache miss occurs for the shared data, if the shared data is kicked out of the cache due to the tasks executed in between the dependent tasks, or due to the task switching mechanism itself. Though, given that the multi-core platforms within PMP contain an abundance of cache compared to the size of data that is shared between dependent tasks, it is safe to assume a cache hit is guaranteed in the majority of these cases. Since the shared interface may consist of multiple layers and protocols, the additional execution time due to dependent tasks assigned to different cores, is, like the intrinsic task weight, a multi-valued metric. Next to the overhead of communicating the shared task data, there is also a penalty to be paid for the dependency resolving mechanism. This penalty should also be taken into account.

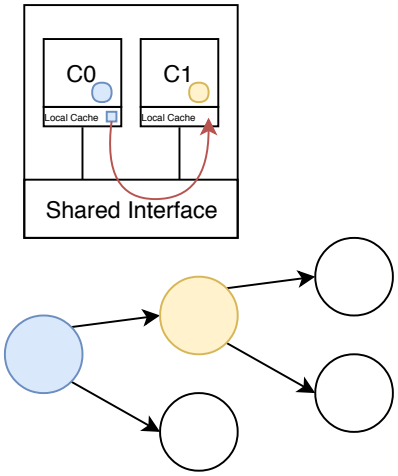


Figure 5.5: Sharing data between tasks.

M3. Instruction Cache Penalty: Another cause for execution time variation are instruction cache misses. As indicated in Chapter 4, within PMP there is quite some implementation duplication across tasks (e.g. tasks calling the same functions). If these tasks are scheduled within a relatively short time period onto the same core, the chances of an instruction cache hit might increase, thus increasing the change of an execution time advantage (see Figure 5.6). As in the previous case of data-cache, the chances of such a hit depends on the caching policy, amount of instruction cache, and all instructions that are executed in between calling the same function. Due to the wide variety in implementations and compiler optimizations, it is expected that identifying instruction re-usage across tasks, is difficult, moreover it is expected that predicting a hit is more difficult compared to the previous case.

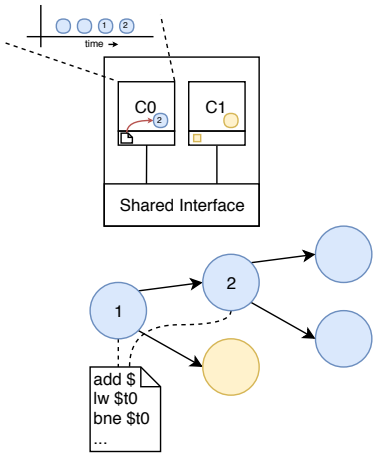


Figure 5.6: Instruction cache re-usage.

M4. Data Cache Penalty for Non-shared Data: Besides a potential data cache hit due to dependent tasks assigned to the same core, a data cache hit for the internal state data of a particular task can also occur. As shown in Chapter 2, the DAGs of tasks which constitute the customer's application, are handled in a periodic fashion, at a certain control frequency. Since the assignation does not change during run-time, a cache hit may occur across cycles. Figure 5.7 visually portrays the potential advantage. The chance of a data cache-hit for the internal state of a particular task t_x, depends on the tasks that follow in between the current and next cycle, after which this task t_x, is executed again. Furthermore the chance of such a data cache-hit strongly depends on cache sizes and caching policies. Because of this, it is expected that measuring, let alone taking into account this potential cause of execution time variation, is both the most difficult, moreover expected to have the least amount of effect on the resulting execution time.

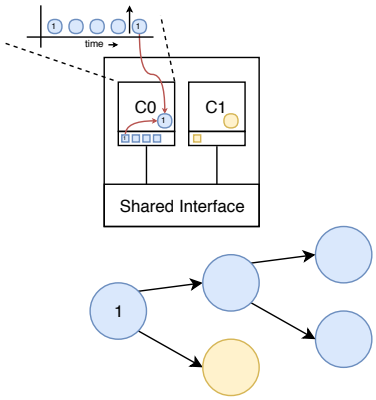


Figure 5.7: Data cache re-usage for internal state data.

5.2.2 Measuring Techniques

In the following sections, some performance measuring techniques will be discussed which can be used to measure the metrics presented in the previous section.

5.2.2.1 Static Measurements

The software implementations of the sensors, actuators, control algorithms, could be analyzed offline, i.e. without actual execution. If Worst Case Execution Timing (WCET) is to be guaranteed, static measurements are well suited since (unlike dynamic measurements), the prediction is never lower than the actual WCET, if carried out correctly. In order to provide the same guarantees using dynamic measurements, either every possible input combination has to be analyzed or the Worst Case input combination has to be known beforehand, which is only possible if Static analysis has been performed.

Assuming that recursion is non-existent and all loops are bounded, moreover presuming that the source code of every task is available (including customer code blocks), all tasks could be compiled to assembly for every hardware platform that is available in PMP. Then all possible branches need to be considered in order to find the branch that determines the Worst Case Execution. If the Worst Case Execution flow is found, the amount of required cycles can be calculated using known values for the cycle count of each instruction, for a specific hardware target. Note that this assumes no instruction cache misses, which may occur in the Worst Case. However assuming an instruction cache miss for every instruction is too stringent. For data read and write instructions, the cycle counts can be used assuming a data cache miss, which again, is quite rigorous, even for the Worst Case. These cases immediately portray the difficulty of determining a tight bound on the Worst Case Execution time of each task, especially given that pipelines, bus contention, branch prediction, etc. are not even considered yet. However, given that all these factors are accounted for, guarantees can be given for the Worst Case Execution time, which is a necessity in hard real-time systems.

Advantages

- Best solution to find bounds for the WCET (or BCET).
- No additional run-time overhead (the measurements can be performed at compile time).

Disadvantages

- Accurate static measurements are generally a lot more difficult than dynamic measurements. However if Worst Case Timing is to be guaranteed (hard real-time systems), static measurements are a must.
- If recursion or loops occur without a bound on the amount of iterations, static measurements are not suitable. (Though these should of course be avoided if a system is hard-real time).
- Approach is not universal, new hardware requires a new framework (different instruction set, different cycle count etc.).

- New implementations or updated implementations require new measurements.
- Either the source code or compiled assembly code of customer blocks need to be available.
- Not really suitable for Average Case Timing Analysis.

Is is expected that static measurements will not be a viable solution to measure task durations within PMP, since PMP focuses on Average Case Execution efficiency rather than Worst Case Execution guarantees. Static timing analysis is ideal for finding (tight) bounds for the Worst Case and even Best Case (e.g. taking the shortest branch). However it is not suited for Average Case timing analysis. For example, static analysis for the Average Case immediately brings some additional difficulties:

- Which branch determines the average case?
- What will the hardware do in the average case? (cache behaviour, branch prediction, pipeline stalls, etc.)

Furthermore generating tight WCET bounds using static analysis is a difficult and widely studied topic, it requires an extensive research and possibly partial system redesign in order to do it properly. Furthermore this research mainly focuses on *average case scheduling*, thus WCET bounds are not necessary.

Even though the static timing analysis ought to be of no use for determining the average task weights, it could be useful in determining the average edge weights. Since edge weights represent the overhead introduced when two dependent tasks are mapped to different cores (which is some kind of worst case, since there is a guaranteed cache miss), static analysis is applicable. If the amount of shared data is known, it is expected that the overhead can be predicted quite accurately.

5.2.2.2 Dynamic Measurements

Another option is to do dynamic measurements, that is, timing measurements when the system is executing some kind of schedule. The main advantage compared to the static measurements is its lower complexity.

A possible design would be a framework that allows to repeatedly run (completely) different schedules. In order to achieve this, the motion platform should be extended with a low-overhead schedule switching mechanism. Furthermore system timers should be used to measure the execution times of each individual task.

Advantages

- Less complex than the static measurements.
- Different execution orders can be measured.
- Cache influences can be measured to some extend.
- Approach is universal (new hardware does not require a new framework).

Disadvantages

- Not suitable for Worst Case Timing (at least not without combining static measurements).

5.2.2.3 Hardware Performance Monitors

It is possible to use hardware performance monitors to measure for example cache influences. Since performance monitoring using hardware performance counters is strongly hardware dependent, support for the hardware targets whereupon PPCx3 and XEONx3 are based, should be verified. In Table 5.1 the support for the hardware targets, whereupon PPCx3 and XEONx3 are based, is shown.

Table 5.1: Hardware support.

Metric:	Freescale E500mc	Intel D-1500
DCache miss-count:	L1 + L2[1]	L2 + L3
ICache miss-count:	L1 + L2[1]	L2 + L3
DCache miss-penalty	n/a[2]	n/a
ICache miss-penalty	n/a[2]	n/a

[1] Using a combination of registers. See table 9-46 in the e500mc reference manual [22].
[2] The E500mc does have some cycle counters for data misses in the Memory Management Unit (MMU) (memory management unit) however it is not clear what misses are exactly counted nor is it clear in which layer. Furthermore there are no performance monitors to measure a specific layer (L1 / L2).

In short, the performance monitors are excellent for measuring cache miss counts, however it is not possible (or straightforward) to measure the penalties that occur purely based on the information that is available in the performance registers.

Advantages

- Best way (and probably the only way) to measure the amount of misses that occurred.

Disadvantages

- Hardware dependent implementation.
- Cache-miss penalties not (directly) measurable.

5.2.2.4 Cache Invalidation

In order to make cache behaviour more predictive, it might be possible to perform cache invalidations on different cache levels. Unfortunately, like the hardware performance monitors, cache invalidation is strongly architecture specific. Moreover cannot be generalized, since each architecture may use different amounts of cache layers, as well as different caching policies. The only meaningful abstraction would be a differentiation between local and shared cache invalidations, and instruction or data cache invalidations,

without considering a cache layer in specific.

Advantages

- Cache misses for every task can be forced which might improve the measurements.

Disadvantages

- Hardware dependent implementation.

5.2.3 Combining Methods and Metrics

In this section the following task DAG (see Figure 5.8) will be used within each example.

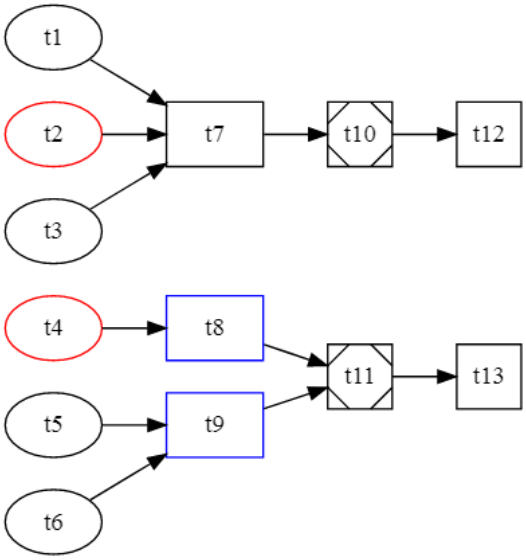


Figure 5.8: Example task DAG.
Nodes with the same shape and colour represents nodes sharing the same implementation.

In the context of dynamic measurements, a certain "test" schedule is represented in the following way:

Table 5.2: Example schedule.

c_0	t_1	t_2	t_3	t_6	t_9	t_{10}	t_{11}	t_{13} D
c_1	t_4	t_5		t_7	t_8		I t_{12}	

The gaps indicate dependencies (e.g. t_3 should finish before t_7), however start times and stall times are not shown. Furthermore the "I" indicates an instruction cache invalidation and "D" indicates a data cache invalidation. Note an invalidation can be performed either before or after a task. Furthermore note that the level that is to be invalidated, will be indicated in each explanation.

5.2.3.1 M1 (Intrinsic Task Weight)

Static Measurements Based on the conclusions drawn in Section 5.2.2.1, it was decided to leave out the static measurements for metric M1.

Dynamic Measurements Dynamic measurements can be used to measure metric M1. Preferably the measurements should be done in such a way that they capture the intrinsic execution variation due to different branches within tasks, without: capturing the variations, due to the used schedule, the dynamic behaviour of the cache, or core-to-core overhead. It is tricky to take all factors into account, however, there may be some possibilities to minimize the effects of the aforementioned influences.

- Using a single-core schedule
- Flushing the cache in between tasks
- Randomization

The first possibility eliminates any core-to-core overhead, which is desired. The only downside is the additional load added to the system. However, during dynamic measurements, it is possible to temporarily reduce the cycle period if necessary. The second possibility is more drastic. It will increase the load significantly (especially in combination with a single core schedule), moreover, it will give an overestimate of the average task execution time. Though, if the total cache penalty for a task can be measured in some way, the averages can be updated by subtracting the expected penalty, if for example a cache hit is to be expected (which can be part of the scheduling algorithm). The main advantage is that, after a cache flush, it is quite certain that a cache miss will occur. Better would be to measure the tasks, having the certainty a cache hit occurs. One could use task duplication to achieve this, however task duplication is not possible within PMP, since it would affect the internal states of each control algorithm.

Instead of flushing the cache, another option is to interpret timing variations due to dynamic cache behaviour in the same way as the variations due to branches within a task. To reduce the chance a particular task has an advantage due to more cache hits compared to other tasks due to its placement within the schedule, a solution would be to create a predefined amount of randomized schedules in order to measure the average task execution times. Furthermore it reduces the chance a task would benefit from other influences like branch miss-predictions, cache snooping, etc.

Given the advantages and disadvantages portrayed above, it was decided that the dynamic measurements for M1 are best performed using single-core randomized schedules. Previous internal research performed at Prodrive showed that cache misses for tasks in between cycles do not occur often (at least not on PPCx3), thus forcing a cache miss will definitely result in over-estimations for the average task execution times. Furthermore support for cache flushes varies per platform, moreover requires hardware-dependent implementations. Therefore it was decided to perform the measurements without cache flushes in between.

An example of a randomized scheduler for the dynamic measurements can be made using the following steps:

1. Make a list L of schedulable nodes which consists of all top nodes
2. At random, pick one of the schedulable nodes $n_x \in L$
3. Remove n_x from L and get all successors $SUCC(n_x)$
4. For all successors $n_y \in SUCC(n_x)$
   (a) If all predecessors $n_z \in PRED(n_y)$ are already scheduled, add n_y to the schedulable list of nodes $L = L \cup n_y$
5. Repeat from step 1 until L is empty $L = \emptyset$

Advantages

- Several repetitions are possible, which (in general) improves the accuracy.

Disadvantages

- Due to all unpredictable influences, the averages found during dynamic measurements could be inaccurate.

5.2.3.2 M2 (Core-to-core Overhead for Shared Data)

Static Measurements Within a PMP product, configurations can be applied by the customer that specify connections between CLCs. These connections are represented by shared signals. The amount of connections, and the size of each input, is known during run-time. If the penalty per unit shared data can be measured once, the expected overhead can be estimated for each edge.

All shared signals have been memory aligned, meaning that any data-type which fits within the cache-line width of the processor would (in theory) require the same amount of time to be fetched. Therefore the cache-line width ought to be a representative base unit for estimating the overhead. Depending on the desired accuracy, a choice can be made to use an average input size for every edge, or to make the overhead input size (and thus edge) dependent.

Apart from the shared data, core-to-core overhead also depends on the efficiency of the dependency resolving mechanism, which is currently a producer and consumer model. The functionality and expected overhead of a producer and consumer does not depend on the configuration, therefore a single estimation or measurement should be enough. The ”real” running time of a consumer depends of course on the placement in the schedule, e.g. if a consumer is executed before the producer the waiting time can be quite long. However the waiting time is not part of the core-to-core overhead, it is actually a hole in the schedule, which (if it occurs), ideally, is a deliberate choice of the scheduler which already took this idle time into account.

Dynamic Measurements In order to measure the core-to-core overhead, the dynamic timing framework can be used in the following way:

Use a (random) single core schedule (see Table 5.3), or use the measurements from metric M1.

Table 5.3: Example single core schedule.

c_0	t_1	t_2	t_3	t_7	t_{10}	t_{12}	t_4	t_8	t_5	t_6	t_9	t_{11}	t_{13}
c_1													

And in the second measurement follow the following steps:

1. Make a list of available Edges L_e
2. For each edge $e_x \in L_e$
   (a) Put the node from which the edge originates $src(e_x)$ including all its predecessors $PRED(src(e_x))$ on core p_0
   (b) Put all other nodes on p_1

In order to demonstrate the flow, a portion of the example graph will be used (see Figure 5.9).

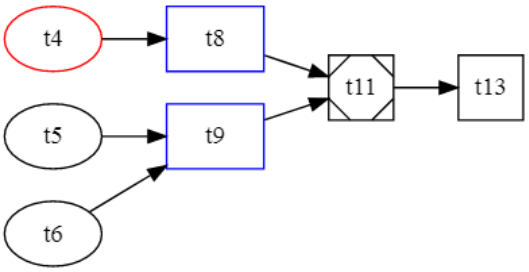


Figure 5.9: Portion of the example task graph.
Nodes with the same shape and colour represents nodes sharing (parts of) their instructions.

First a list is constructed of all available edges: $[(t4 \rightarrow t8), (t5 \rightarrow t9), (t6 \rightarrow t9), (t8 \rightarrow t11), (t9 \rightarrow t11), (t11 \rightarrow t13)]$. The next step is to create a schedule for each edge that allows to measure the overhead induced, which is presented in Tables 5.4 up till and including 5.9.

Table 5.4: Example measuring schedule for $(t_4 \rightarrow t_8)$.

c_0	t_4						
c_1		t_8	t_5	t_6	t_9	t_{11}	t_{13}

Table 5.5: Example measuring schedule for $(t_5 \rightarrow t_9)$.

c_0	t_5					
c_1	t_6	t_9	t_4	t_8	t_{11}	t_{13}

Table 5.6: Example measuring schedule for $(t_6 \rightarrow t_9)$.

c_0	t_6					
c_1	t_5	t_9	t_4	t_8	t_{11}	t_{13}

Table 5.7: Example measuring schedule for $(t_8 \rightarrow t_{11})$.

c_0	t_4	t_8			
c_1	t_5	t_6	t_9	t_{11}	t_{13}

Table 5.8: Example measuring schedule for $(t_9 \rightarrow t_{11})$.

c_0	t_5	t_6	t_9		
c_1	t_4	t_8		t_{11}	t_{13}

Table 5.9: Example measuring schedule for $(t_{11} \rightarrow t_{13})$.

c_0	t_4	t_8	t_5	t_6	t_9	t_{11}	
c_1							t_{13}

Based on the desired accuracy and complexity of the scheduling algorithm, either the average values can be measured or a distribution.

The idea behind the measurements is as follows: If for example t_{13} and t_{11} were to share data, the execution time of t_{11} for the $(t_{11} \rightarrow t_{13})$ measurement, should on average be higher than the single core measurement, because the shared data has to either be communicated (e.g. via cache snooping), or retrieved all the way from the shared memory interface. The same holds for all other core-to-core edges. By repeating the measurements, moreover averaging all the values an indication of the core-to-core overhead can be extracted.

The aforementioned method does not take into account that, tasks sharing the same code implementation could in theory have an additional disadvantage when scheduled onto different cores. In order to prevent this, instruction cache invalidations could be performed between tasks for all schedules depicted above. Furthermore the data cache could be invalidated at the start of each cycle, in order to exclude any benefits for non-shared data re-usage (e.g. internal state data). The data cache should only be invalidated at the start, because the advantage of executing two dependent tasks, say t_3 and t_7, on the same core lies in the fact that the shared portion of the data is already in cache. If during these measurements the data cache would have been invalidated, the single core dependency advantage would disappear. Moreover a single data cache invalidation at the start of each cycle should already introduce data cache misses for all non-shared data of each task.

Since these invalidations can be performed on both the single and multi-core measurements, the timing difference between these values should in theory more accurately depict the real core-to-core overhead compared to the “invalidation-less” case, because the metric to be measured, is better isolated. Unfortunately the hardware support for cache

invalidations differs between architectures, thus a solution including cache invalidations, would not be applicable in a generic way.

There are also corner cases in which the method described above cannot properly isolate the overhead measurement of a particular edge. One of the cases is when there exist a "redundant dependency" in the graph, which actually occurs in one of the products (see Section 4.2). Given the example graph in Figure 5.10, if edge $(t_9 \rightarrow t_{11})$ is to be measured using the method described above, nodes t_5, t_6, and t_9 will be placed on core A and all other nodes on core B. However, this means that during the edge measurement of $(t_9 \rightarrow t_{11})$, edge $(t_6 \rightarrow t_{11})$ also has influence on the measurement. A solution would be to add all predecessors of the source node from which the edge originates (t_9), that also happens to be a direct predecessor of the destination node (t_{11}), to core B. This however makes the measurement method a bit more complicated, since (depending on the amount of redundant edges) multiple dependency resolving guards have to be inserted.

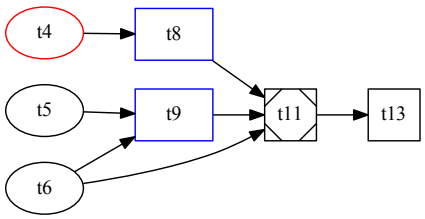


Figure 5.10: Example task graph containing a "redundant" edge.

Since the schedule is to be executed using the dynamic measurements, producer and consumers should be added to make sure the dependencies are met. These producers and consumers can be timed with a separate timer. The idea is to extract the average overhead introduced by a producer and consumer model in the best-case, that is, the overhead given that the consumer can pass immediately, and add it to each edge weight. Measuring the producer and consumer overhead can be done offline and has to be performed only once, since it is not affected by the configuration.

Advantages

- The measurements are simple and expected to be quite representative.

Disadvantages

- Cache invalidation is not possible on all available hardware targets.

5.2.3.3 M3 (Instruction Cache Penalty)

Dynamic Measurements To measure the average penalty induced by an instruction cache miss using dynamic measurements, an option is to: both execute a schedule with instruction cache invalidations in between and the same schedule without these invalidations, repeat the measurements for several iterations, and calculate the execution

time difference. Assuming data cache behaviour is left unaffected by the instruction cache invalidations, any execution time advantage ought to be caused by instruction cache re-usage for the majority of cases.

In order to minimize any potential data cache influences, a possible optimization would be to introduce data cache misses in between each task for both measured schedules.

Note that these measurements are not possible on all available hardware targets, moreover it has to be specified in which layer a cache miss should occur. Furthermore, these measurements are only useful if, after a cycle, there is a significant chance that an instruction cache miss occurs, meaning that within the cycle, it is beneficial to schedule tasks having the same implementation onto the same core. If after a complete cycle, instruction cache misses do not occur often, the benefits of scheduling tasks onto the same core with respects to instruction cache re-usage diminishes.

Advantages

- Additional information (if accurate) generally means more scheduling possibilities.

Disadvantages

- Does not work for all available hardware platforms.
- Requires a cache-aware scheduling algorithm in order to take advantage of the measurements.

5.2.3.4 M4 (Data Cache Penalty for Non-shared Data)

As mentioned in section 5.2.1, M4 is both the most complex and the least relevant of all metrics. Therefore it was decided to leave it out of scope.

5.2.4 Implemented Designs

During the iterative implementation approach, the Dynamic Measurement design was implemented. As described in Section 5.2, Dynamic Measurements allow to repeatedly run (completely) different schedules and measure the individual task execution times. In order to support this measurement technique, the existing online scheduler (depicted in Figure 5.2) was extended such that schedules could be changed during run-time, moreover, execution times could be measured per task.

5.2.4.1 Dynamic Node Weight Measurements

In order to leave out core-to-core dependency influences, task time measuring is performed using a single core schedule. In order to create a single core schedule a scheduler solution was added to the ESL which follows the following steps:

1. Create a list of schedulable nodes L_S and a list of schedulable candidates L_C.
2. Retrieve all top nodes from the DAG and add them to the list of schedulable nodes $L_S = T_g$.
3. While $L_S \neq \emptyset$

(a) Pop the first node n_1 in L_S and add it to the first core in the schedule $S[c_1]$.

(b) Add all successors of this node $SUCC(n_1)$ to the schedulable candidates list $L_C = L_C \cup SUCC(n_1)$.

(c) For all nodes n_x in L_C

i. Get all direct predecessors $PRED(n_x)$

ii. For all nodes n_y in $PRED(n_x)$

- If all direct predecessors are already scheduled ($n_y \in S$), add n_y to the schedulable nodes $L_S = L_S \cup n_y$
- Remove n_y from L_C

Using these steps, a single core schedule is created that does not violate any dependencies. After the single core schedule is created, the schedule is dynamically updated in the RTC. After the schedule update has been performed, the sampling process starts with a user defined amount of iterations. Note that the randomized schedule design has not been utilized, since it would have increased the amount of measurement iterations and thus increase the total scheduling time.

5.2.4.2 Dynamic Edge Weight Measurements

In order to measure edge weights, the design described in Section 5.2.3.2 was implemented. Within the Dynamic Edge Measurements, all edges are retrieved from the DAG. Then, for each edge, an isolated schedule is created and the execution time of the destination node is compared against the result found during the Dynamic Node Measurements. The steps of the Dynamic Edge Measurements routine are shown below:

1. Input: edge to isolate e_x.
2. Create a list of unscheduled nodes L_u.
3. Get all (direct and indirect) predecessors of the source node $P_s = ALLPRED(src(e_x))$.
4. Repeat for the destination node $P_d = ALLPRED(dst(e_x))$.
5. Remove the source $src(e_x)$ and destination $dst(e_x)$ nodes from both predecessor lists (P_s and P_d).
6. Store the symmetric difference of P_s and P_d. $Diff_{sd} = P_s \triangle P_d$.
7. Create a single core schedule $S[c_0]$ for P_s on core c_0, using the single core scheduler described earlier.
8. Add the source node $src(e_x)$ to the end of this schedule $S[c_0]$.
9. Reserve a guard pair (producer and consumer).
10. Schedule the producer to the end of the schedule $S[c_0]$ on core c_0.
11. Schedule the consumer to the schedule $S[c_1]$ of core c_1.

12. Create a single core schedule $S[c_1]$ for $Diff_{sd}$ on core c_1. This makes sure that any predecessor of $dst(e_x)$ that is not a predecessor of $src(e_x)$ is scheduled before $dst(e_x)$, in order not to violate any dependencies.
13. Add the destination node $dst(e_x)$ to the end of this schedule $S[c_1]$.
14. Update the unscheduled list L_u by removing the scheduled nodes .
15. Create a single core schedule $S[c_1]$ for the remaining unscheduled nodes L_u on core c_1.

Figure A.4 in Appendix A.5 presents two examples of isolated edge measurement iterations in PPCx3.

5.2.4.3 Dynamic Producer Consumer Measurements

Apart from the (dynamic) edge measurement results found using the method described above, also a static part has to be added to each edge weight, due to the static overhead generated by the producer and consumer pairs. As described in Section 5.2.3.2 the idle period within the consumer and producer model, caused when a producer precedes the consumer, should not occur in the measurement, since this idle period can only be caused by holes in the schedule. Therefore, to measure only the static overhead of a producer consumer pair, a guard isolating scheduler has been implemented and added to the ESL which follows the following steps:

1. Reserve a guard pair GP_0.
2. Reserve another guard pair GP_1.
3. Schedule the producer $prod(GP_0)$ of GP_0 on the first core $S[C_0]$.
4. Schedule the producer $prod(GP_1)$ of GP_1 to $S[C_0]$.
5. Schedule the consumer $cons(GP_1)$ of GP_1 on the second core $S[C_1]$.
6. Schedule the consumer $cons(GP_0)$ of GP_0 to $S[C_1]$.

In this way, it is assured that the execution of consumer $cons(GP_0)$ will precede the execution of the corresponding producer $prod(GP_0)$ assigned to the other core, preventing any idle period. The combination of the execution time measurements of both consumer $cons(GP_0)$ and producer $prod(GP_0)$ is used as the static part of the edge weights and is added to all the edge weights found using the edge isolating measurement method.

5.3 Scheduler Framework

It was decided to make a modular design for the scheduler framework. Many assignation and scheduling concepts presented in Chapter (3.2) can be classified as improvements that can (in theory) be used by any of the schedulers. In combination with the Extendable Scheduler Library as described in Section 5.1, a modular design would allow to combine different schedulers and improvements, which may lead to better solutions. Furthermore,

many schedulers share the same concepts, for example the calculation of bottom and top-levels, which can be reused within the modular design.

Besides some of the algorithms presented in Chapter 3, during the iterative implementation phase, a Load Balancing Processor Assignment extension for Sarkar's Internalization algorithm was designed and implemented, based on the experimental results found during the iterative approach. In addition to this extension, an iterative approach was designed, which specifically targets disjoint sub-graph structures, which, as presented in Chapter 4, can be found in the DAG of PPCx3. In this section, the designs of both methods are described.

5.3.1 Load Balancing Processor Assignment

Since Sarkar's processor assignment, as described in Section 3.2, appeared to perform worse than expected, a more simplistic processor assignment approach was designed. The *Load Balancing PA* algorithm uses Round Robin / Simple Load Balancing to equally divide the clusters created by a clustering algorithm, over the available hardware resources. In order to take into account edge weights, the algorithm first sorts all nodes by their (dynamic) bottom-level, which is determined in the same way as the *Internalization* clustering algorithm described in Section 3.2.3.1. For each node in the sorted list, the cluster weight of the cluster in which this node resides, is determined; after which all nodes within this cluster are scheduled to the processor having the lowest accumulated cluster weight. The Load Balancing PA algorithm is explained using the following steps:

1. For each virtual cluster $c_x \in C_{virt}$, accumulate the weight of all nodes inside the cluster, and store its value $W_{c_x} = \sum W_{n_x} \forall n_x \in c_x$.
2. Next to the virtual cluster mapping, create a mapping from physical clusters C_{phy} to weight value. Zero initialize all entries.
3. Create an unscheduled list of nodes sorted by bottom-level $L_{unscheduled}$.
4. While $L_{unscheduled} \neq \emptyset$:
   (a) Pop the first node from the unscheduled list $n_0 = L_{unscheduled}[0]$, which is the currently unscheduled node having the highest bottom-level.
   (b) Find the virtual cluster $c_{cid(n_0)}$ in which n_0 resides.
   (c) Initialize a variable called *load* and assign it its maximum value.
   (d) For each physical cluster $c_y \in C_{phy}$:
      i. Add the virtual cluster weight to the physical cluster weight $weight = W_{c_{cid(n_0)}} + W_{c_y}$.
      ii. if $weight < load$:
         A. Update the (winning) cluster $c_{winner} = c_y$.
   (e) Move all nodes from the virtual cluster c_y to the winning physical cluster c_{winner}.
   (f) Remove all nodes inside c_{winner} from the unscheduled list.
   (g) Update the weight value of the winning cluster $W_{c_y} = W_{c_y} + W_{c_{cid(n_0)}}$.

5. For each physical cluster $c_y \in C_{phy}$:

   (a) Order nodes in the cluster using the same steps as in the Internalization algorithm.

   (b) Schedule the nodes inside the cluster to the corresponding hardware resources.

5.3.2 Dynamic Cluster Splitting

In Section 4.1 it was demonstrated that PPCx3 contains a lot of independent sub-graphs within the complete calculate DAG. Since these sub-graphs are independent, each of these sub-graphs is an ideal candidate to be parallelized. All the aforementioned schedulers however, do not explicitly take these sub-graphs into account and often (unnecessary) break up these sub-graphs. Therefore a new implementation is proposed called *Dynamic Cluster Splitting*, which, instead of merging clusters, breaks up clusters if the parallel time decreases. The main idea of *DCS*, is to create clusters or *Super Nodes* of each sub-graph, then try and schedule this so called *Super Graph*. After the parallel time is measured, using some strategy, these *Super Nodes* can be split up after which the parallel time is measured again. This splitting process can be repeated as long as the measured parallel time keeps decreasing. The main idea behind this algorithm is that, if these "ideal" parallelizable sub-graphs happen to divide well over the available hardware resources, why bother try and splitting them up. If these parallelizable sub-graphs do not divide well, the *DCS* algorithm will split some of the nodes as longs as the parallel time decreases.

5.3.2.1 Measuring Instead of Computing

Using dynamic measurements, it is possible to do a parallel time measurement in between each iteration of the *DCS* scheduler. Measuring the parallel time instead of calculating it using node weights, is more accurate, since the measured node weights do not always represent the execution time in a specific schedule and assignation.

5.3.2.2 Sequential Splitting

The first proposed splitting strategy is *Sequential Splitting* in which a *Super Node* is split "Sequentially", that is, in each iteration, split the bottom-nodes from the other nodes in the *Super Node*'s graph, and keep continuing until the accumulated weight difference between both sub-sets of nodes is as small as possible. After the split point, and thus two sub-sets of nodes are determined, created two *Super Node*'s (s_0 and s_1) having a single edge in between such that no dependencies are violated. Note that this dependency may be quite coarse compared to the underlying dependencies, however, this could be advantageous cause now only a single producer and consumer is required to resolve the dependency (if scheduled to different cores). This method is called sequential splitting because dividing the newly created *Super Nodes* (s_0 and s_1) over two cores, will always be slower than sequential execution on a single core, since execution of s_1 cannot commence before all nodes in s_0 have been executed. However, because of the sequential split, there

is more freedom in placement on a single core. It is for example possible to execute s_0, another *Super Node* in between, followed by s_1.

5.3.2.3 Parallel Splitting

The second proposed strategy is *Parallel splitting* in which a *Super Node* is split in a parallel fashion. The idea is to identify all split/fork and merge/join nodes in the graph. Each disjoint path between a split and merge node, or split and end node, should in theory parallelize well given that the paths are of similar weight. A split node is defined as a node with two or more outgoing edges, a merge node is a node having two or more incoming edges. A node can also be both a split and a merge node at the same time. Figure 5.11 depicts an example graph were all split (S) and merge (M) nodes are identified.

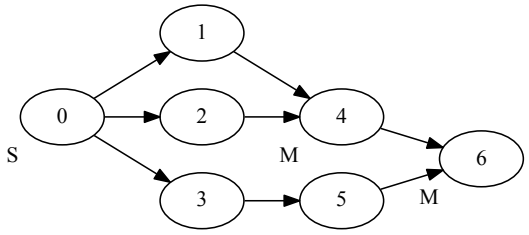


Figure 5.11: Split and merge nodes example.

Using the parallel split method, the *Super Node* graph example found in Figure 5.11 can be split in several ways. For example a possible split would be nodes (n_0, n_1, n_2, n_4) and (n_3, n_5, n_6). In which it is also possible to include n_0 in the other node set or n_6 vice versa. Or split between split node n_0 and merge node $n4$ instead of n_0 and n_6. Within the split routine, the combination of parallel paths should be found in which the weight difference between the paths is as small as possible. Note that in contrast to the Sequential Splitting method, the Parallel Splitting method may split a Super Node in more than two new Super Nodes, which depends on the amount of parallel paths that can be found.

5.3.2.4 Exposing Super Nodes

Another strategy is the more simplistic approach, that is, exposing all nodes within a *Super Node*, thus basically destroying the *Super Node* itself. Using this strategy, the "top-level" scheduler (which schedules the *Super Graph*) itself is given scheduling freedom of all nodes in the exposed *Super Node*.

5.3.2.5 Algorithm Steps

The *DCS* scheduler follows the following steps:

1. Construct a *Super Graph* by identifying each sub-graph in the main DAG and creating *Super Node*s of these sub-graphs.
2. Use the internal scheduler to construct a schedule.
3. Use the performance measurement framework to extract the initial Parallel Time $pt_{current}$.
4. Order all nodes in the *Super Graph* by node weight and put them into a list L_n.
5. For all nodes in this list $n_x \in L_n$:
   (a) If the node is a *Super Node* and the amount of nodes is higher than 1:
      i. Use on of the*Super Node* splitting techniques.
      ii. Using the internal scheduler, schedule the *Super Graph* again which internally calls each single core scheduler contained within each *Super Node*.
      iii. Perform a Dynamic Custom Measurement to extract the new Parallel Time pt_{new}, furthermore update the weights of each *Super Node*.
      iv. If $pt_{new} < pt_{current}$ then update the value $pt_{current} = pt_{new}$.
      v. Else Terminate.

To identify each subgraph and create a *Super Graph*, a fill-based graph traversing algorithm was applied. The steps are shown below:

1. Get a list of undiscovered nodes containing all bottom nodes $L_{undiscovered}$.
2. Initialize a list of subgraph nodes lists $L_{subgraphs}$.
3. While $L_{undiscovered} \neq \emptyset$:
   (a) Pop the first node in the list $n_x = L_{undiscovered}[0]$.
   (b) Get a list of all direct and indirectly connected nodes $L_{conn} = GetConnected(n_x)$ and add this list to to the list of sub-graph node lists $L_{subgraphs}$.
   (c) Remove all nodes that resides in both the undiscovered and connected list from the undiscovered list, i.e. remove the intersection $L_{undiscovered} \cap L_{conn}$ from $L_{undiscovered}$. This is mandatory since a sub-graph might include several bottom nodes.
4. For all subgraph node lists L_{sub} in the list of subgraph nodes list: $L_{sub} \in L_{subgraphs}$
   (a) Create a subgraph of the complete graph using the nodes in L_{sub} thereby adding all the required edges from the complete graph to the subgraph.
   (b) Create a *Super Node* and add the newly created subgraph.
   (c) Add the new *Super Node* to the *Super Graph*.

In order to find all direct and indirectly connected nodes of a node n_x the following steps are performed:

1. Input: n_x.

2. Create an empty node list $L_{connected}$.
3. Call the method $GetAllConnected(L_{connected}, n_x)$ whilst supplying the empty list and n_x.
4. Output: $L_{connected}$.

The *GetAllConnected()* method follows the following steps:

1. Input: n_x, $L_{connected}$.
2. Add n_x to $L_{connected}$.
3. Get all direct neighbors of n_x, that is, all direct predecessors and successors $GetDirectNeighbors(n_x)$.
4. For all direct neighbors $n_y \in GetDirectNeighbors(n_x)$:
   (a) If not already processed $n_y \notin L_{connected}$:
      i. Do a recursive call to find the connected nodes to which n_y connects $GetAllConnected(L_{connected}, n_y)$.
5. Output: $L_{connected}$.

5.3.3 Transitive Reduction

During the pre-analysis phase it was observed that redundant edges may be introduced within the task graph (see Section 4.2). These redundant edges makes the scheduling process more difficult, furthermore adds up to the total time it takes to produce a schedule. A possible solution is to remove these redundant edges, which is better known as transitive reduction. However it could also worsen the performance of the schedule because, even though the edge is redundant, there is still data to be shared between the tasks connected to this edge, which is likely to introduce an additional penalty if these tasks are mapped to different processing units. Transitive reduction is applied using the following steps:

1. For all edges in the graph $e_x \in E$:
   (a) Get the predecessors of the destination node $PRED(dst(e_x))$.
   (b) For all predecessors $n_x \in PRED(dst(e_x))$:
      i. If n_x is a descendant of the source node $src(e_x)$, remove the edge e_x from the graph.

To check if a node n_x is a descendant of another node n_y the following steps are followed:

1. Input: n_x and n_y.
2. If $n_x = n_y$ return *true* and *terminate*.
3. Else get all direct and indirect predecessors of n_x.
4. If n_y appears in this predecessor list return *true* and *terminate*.
5. Return *false* and *terminate*.

5.3.4 Implemented Designs

Besides the transitive reduction implementation, during the iterative implementation phase, the following schedulers were implemented and added to the ESL:

- Original *Mapped* and *Non-mapped* schedulers
- Single Core (used within Dynamic Measurements)
- Edge Isolated (used within Dynamic Measurements)
- Guard Isolated (used within Dynamic Measurements)
- HLFET
- ISH
- Internalization using Sarkar's PA
- Internalization using Load Balancing PA
- DAC

5.4 Conclusion

In this chapter the design for the automated framework was presented. An overview of the complete design was given, as well as the designs for both the scheduling and performance measuring frameworks. In addition an overview was given of all designs that were eventually implemented. In the next chapter, the results are presented for the measuring framework, furthermore all implemented schedulers within the ESL are compared against each other, which includes the original *Mapped* and *Non-mapped* algorithms.

6 Experimental Results

In this chapter, a brief overview is given of the implemented solutions that were described in the previous chapter. In between implementation steps, intermediate results are presented which (among other things) have been used to decided upon the next step in the iterative implementation process. Last but not least, a complete overview is given in which all results are compared against each other.

6.1 Dynamic Measurements

Figure A.2 in Appendix A.4 presents the weights found using all dynamic measurement methods combined for PPCx3 and Figure A.3 in Appendix A.4 presents the graph for XEONx3. It is shown that the core-to-core penalty is quite significant compared to the weight of each node in the graph. Moreover it is shown that sensors require the least amount of computational time, however, are often paired with relatively high connected edge weights. As already indicated in Section 5.2.2.2, the method of dynamic edge weight measuring is not able to isolate cache influences, which is expected to be reflected by the edge weights that are found.

It is expected that edges between similar components should have a similar edge weight (e.g. sensor to control network, or sensor to matrix). Analyzing the figures, it is clear that the expectation holds for the bulk part of both graphs. All edges that ought to have a similar weight, are within the same order of magnitude. There do exists some outliers though, which can either be explained due to the aforementioned shortcomings of the edge measurement method or simply because there is more data to share on that edge. After analyzing the implementation, it was shown that the latter was not the case. Because of this, a more simplistic static edge measurement method, might do a similar or even better job at determining relevant edge weights. Another option would be to average out the results found during the dynamic edge measurements for each of these similar edges, however, this requires a method that can (accurately) identify these "similar edges" in such a way that it also applies to any future PMP products, which may or may not contain these edges.

6.2 Comparing Schedulers

In order to compare each implementation against the original scheduler, the average task execution time for both phases (calculate and prepare) is measured using the same methodology as described in the analysis (Chapter 4). The analysis was performed at an early stage of research, since then, the measurement methodology has been improved. Moreover as mentioned in Chapter 5, PMP itself is an ever ongoing development process.

Because of this, the results presented in this chapter for the original solution, differs from the results found during the analysis. In order to allow for a fair comparison between all algorithms (which includes the original solution), all results presented in this chapter have been gather whilst using the exact same state of development. As presented in Chapter 4, the prepare and calculate task-sets can be seen as two individual sets, divided by a multi-core synchronization point. In order to measure performance, first the parallel time of both task-sets is calculated (which is determined by the highest occurring total execution time amongst all cores), then the sum of both parallel times found, is used as the main performance metric.

6.3 Highest Level First with Estimated Times

The *HLFET* algorithm has been implemented according to the steps described in Section 3.2.1.2. After finishing the *HLFET* implementation it was compared against the original scheduling algorithm. The timing results for the calculate and prepare DAGs are presented in Figure 6.1. As expected, the *HLFET* algorithm does not perform well for the calculate tasks, because it does not take into account the edge weights. As a consequence, many core-to-core dependencies are introduced, which are resolved using the producer and consumer execution guard model. The graph clearly shows a significant amount of red bars, which indicate long idle periods caused by the introduced execution guards.

For the prepare tasks it was expected that *HLFET* would probably perform well, since the prepare DAG is almost free of any dependencies. Unfortunately, the *HLFET* algorithm still manages to introduce a core-to-core dependency, which introduces unnecessary idle times. Interestingly the *Non-mapped* scheduler seems to outperform in both phases, thus it seems that the *Non-mapped* scheduler is better than the *HLFET* scheduler for scheduling the prepare tasks. However, the *Non-mapped* round-robin scheduler has a major disadvantage compared to *HLFET*. Coincidentally the division of tasks by sub-controller, assuming static data re-usage moreover assuming each sub-controller task-set requires an equal amount of computation time, works well for the PPCx3 product. However, this may not be the case for other (new) products. If for example the task-sets belonging to the sub-controllers have significant differences in terms of computational complexity, it is expected that the *HLFET* scheduler -which does take into account the computational load- will do a better job scheduling all tasks.

As mentioned earlier the execution time of both the prepare and calculate phases were added for the *Non-mapped* ($\sim 21.63\mu s$) and *Mapped* variant ($\sim 21.21\mu s$) and compared against *HLFET* ($\sim 22.84\mu s$). Using this metric it is shown that *HLFET* performs $\sim 7.7\%$ worse than the mapped variant and $\sim 5.6\%$ worse than the non-mapped variant. The results can also be found in table form (see Table 6.1).

Table 6.1: *HLFET* relative performance for PPCx3.

	Non-mapped	*Mapped*
HLFET	−5.64%	−7.70%

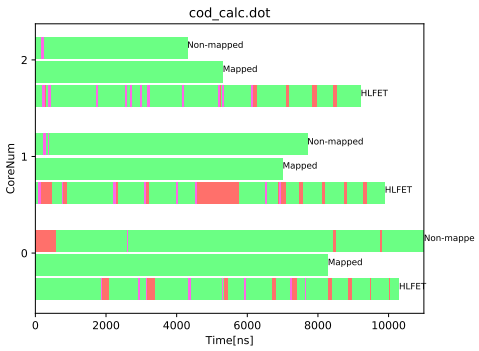
The *HLFET* scheduler has also been tested on the XEONx3 platform (see Figure

6.3). The XEONx3 platform contains a lot more tasks, and a completely different graph structure compared to PPCx3. Unfortunately XEONx3 has no hardware drives connected, thus the original scheduler maps all the tasks to the first core in the system. The *Mapped* variant has been left out of the comparison since it produces the same results as the *Non-mapped* variant, because no configuration file has been created for XEONx3 yet, besides, a configuration file cannot be created anyway, since there are no drives (and thus no sub-controllers), that can be assigned. Because of this, it is less interesting to compare the performance of the original schedulers to *HLFET*. It does however clearly show the shortcomings of the original scheduling implementations. The results are presented in Figure 6.3a and 6.3b. Furthermore, in Table 6.2, the relative percentages can be found.

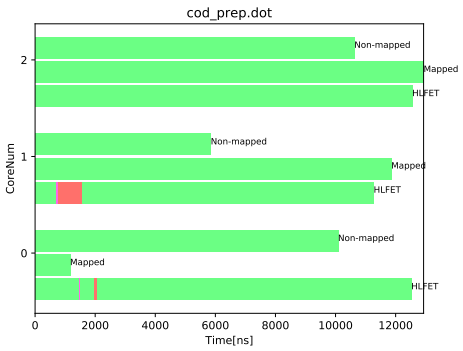
Table 6.2: *HLFET* relative performance for XEONx3.

	Non-mapped
HLFET	34.30%

Last but not least, the total time for each scheduler to produce a schedule has been measured. On average, *HLFET* takes less than a second to schedule both graphs on PPCx3. On XEONx3, *HLFET* requires 39 seconds to produce a valid schedule. Both well beneath the one minute requirement.



(a) Average Calculate task execution timing diagram.



(b) Average Prepare task execution timing diagram.

Figure 6.1: CLC task execution timing diagrams for PPCx3 using *HLFET*

Light-green bars represent *Sensor tasks*, blue: *Actuator tasks*, green: *Other tasks*, purple: *Producers*, light-red: *Consumers*, dark-red: *preceding tasks*.

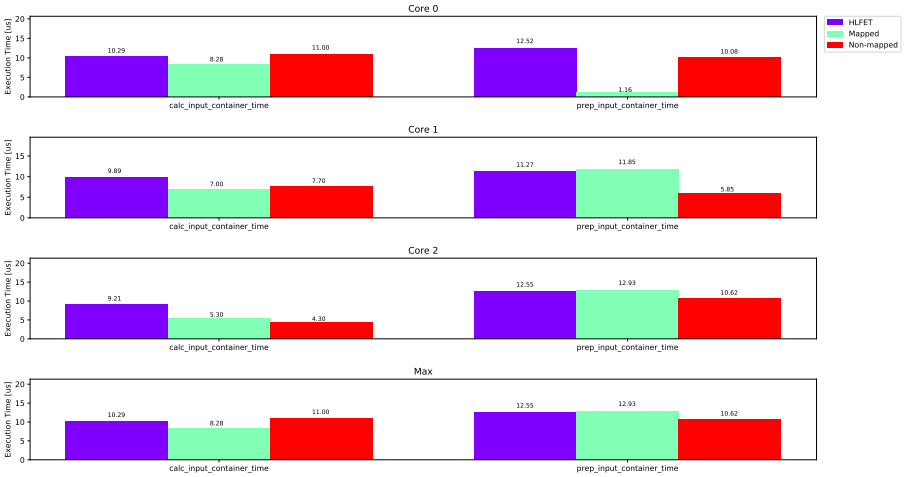
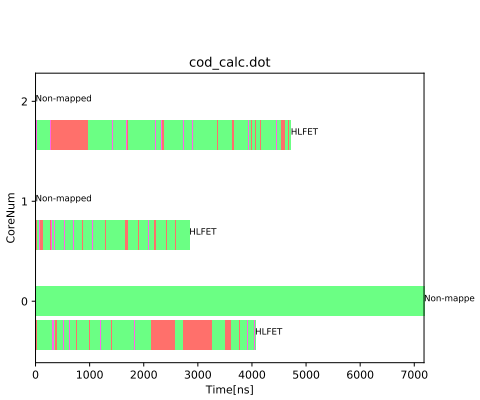
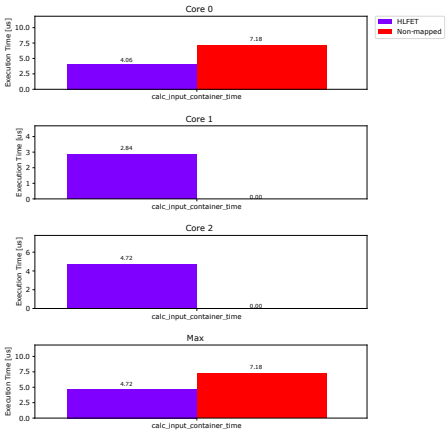


Figure 6.2: PPCx3 execution time comparison using *HLFET*.

The bottom subgraph labeled “max” shows the maximum execution time value over all cores which indicates the total time it takes to finish the execution of the schedules.



(a) Average Calculate task execution timing diagram.



(b) XEONx3 execution time comparison using *HLFET*.

Figure 6.3: XEONx3 results.

See Figures 6.1 and 6.2 for the colour descriptions.

6.4 Internalization

The *Internalization* clustering algorithm is the first clustering algorithm that has been implemented. The algorithm has been implemented according to the steps presented in Section 3.2.3.1.

Figure A.5 in Appendix A.6 show some clustering iterations of the *Internalization* clustering algorithm in PPCx3. The figure depicts that, as expected, *Internalization* clustering starts of by creating clusters for each node in the graph and ends with a

clustered graph assuming an unlimited amount of processing resources.

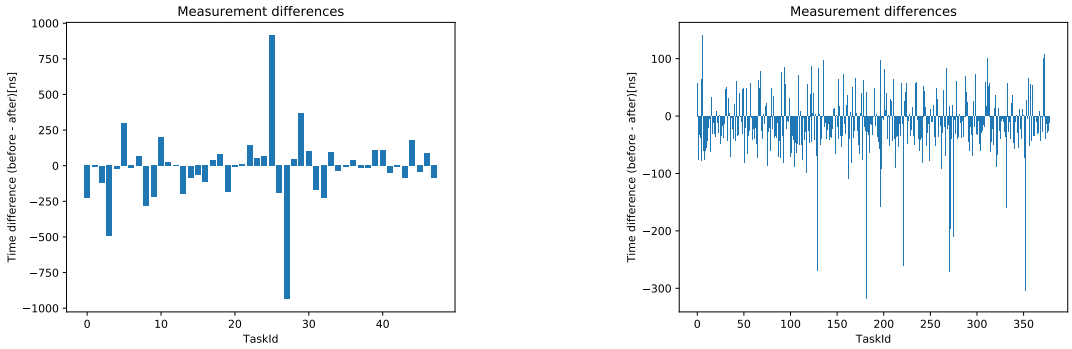
6.4.1 Sarkar's Processor Assignment

The clustering algorithm described above creates a clustered graph assuming an unbounded number of processing resources. To map the clusters to the available physical resources, a so called Processor Assignment algorithm is required. The first Processor Assignment (PA) algorithm that was implemented is Sarkar's Processor Assignment algorithm.

Figure 6.5 and 6.6 present the performance compared to the original scheduler, using *Internalization* including *Sarkar's Processor Assignment* algorithm. Compared to *HLFET*, *Internalization using Sarkar PA* introduces far less core-to-core dependencies and thus fewer guards are added to the schedule. Unfortunately on PPCx3, some of these core-to-core dependencies are placed in an unfortunate position, which introduces significant idle times on both cores c_1 and c_2. However, since core c_0 still finished last, it does not affect the parallel time.

The introduced idle periods can either have been deliberately introduced by the scheduler, or the execution times show significant deviations from the measured values, in which the scheduler could not have made the right decisions. In order to find out, the measured node weights during scheduling were compared against the measured node execution time during system operation. The results are presented in Figure 6.4.

Given the relatively low timer precision of PPCx3 ($\sim 85.3ns$), the bulk of all differences found in the prepare phase are negligible, since the absolute value of these differences are lower than the precision of the timer. There do exist some outliers though, where the final average execution time is higher than was measured during dynamic measurements, however, these variations are expected since changing a schedule can have a significant impact on cache (re)-usage, which in turn affects the execution time. Within the calculate phase, the differences are on average a bit higher compared to the prepare phase. Furthermore two large peaks can be observed in which the expected execution time does not at all reflect the execution time within the schedule. Fortunately these large peaks do not occur often, unfortunately it is difficult to take these peaks into account. Significant execution time differences can occur when sub-controller tasks sharing some static data with non-sub-controller related tasks, which are scheduled onto different cores. However, as indicated in Chapter 4, the non-sub-controller related tasks are static and cannot be dynamically scheduled or assigned.



(a) Differences in the calculate phase.

(b) Differences in the preparation phase.

Figure 6.4: Differences in measured average execution times found during dynamic measurements and extracted after scheduling in the PPCx3 product.

In contrast to PPCx3, *Internalization using Sarkar PA* performs quite well in the XEONx3 product. Observing Figure 6.7a it is shown that, even though XEONx3 contains a lot more edges, *Internalization using Sarkar PA* introduces relatively short amounts of idle periods.

Compared to *HLFET*, *Internalization using Sarkar PA* performs slightly better for both DAGs. Adding up both the execution time of the prepare and calculate phases, *Internalization using Sarkar PA* performs $\sim 4.3\%$ better than *HLFET*, $\sim 3.1\%$ worse than the *Mapped* variant and $\sim 1.1\%$ worse than the *Non-mapped* variant for PPCx3. In XEONx3, *Internalization using Sarkar PA* performs $\sim 23\%$ better compared to *HLFET*, which is a significant improvement.

Table 6.3: *Internalization using Sarkar PA* relative performance for PPCx3.

	Non-mapped	*Mapped*	*HLFET*
Internalization using Sarkar PA	−1.11%	−3.08%	4.29%

Table 6.4: *Internalization using Sarkar PA* relative performance for XEONx3.

	Non-mapped	*HLFET*
Internalization using Sarkar PA	49.61%	23.31%

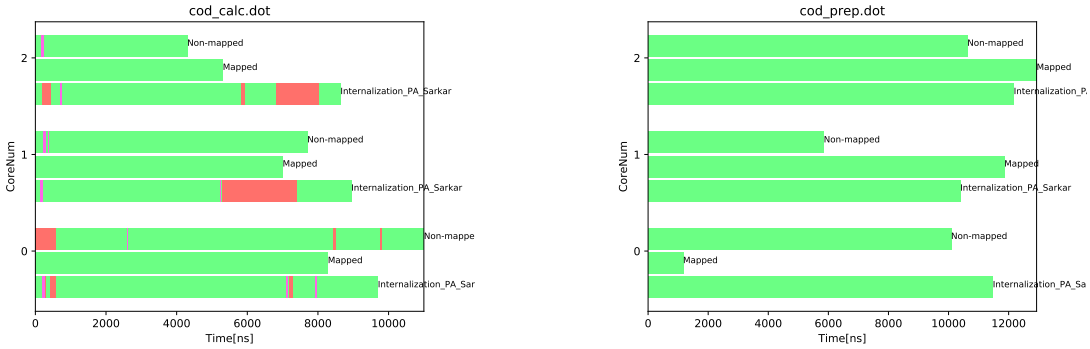
On average *Internalization using Sarkar PA* takes less than a second to schedule and assign the calculate DAG, and 7 seconds to schedule the prepare DAG on PPCx3. On XEONx3, *Internalization using Sarkar PA* requires 200 seconds to produce a valid schedule, which is above the 1 minute requirement.

Internalization using Sarkar PA shows a lot of variation in producing a valid schedule, values twice as high as the average value, moreover ten times as low as the average value, are no exception. After timing analysis, it was shown that the major bottleneck of the algorithm is the introduction of many virtual edges after clusters are merged, especially

at the end of the algorithm where clusters are of significant sizes. Merging these relatively large clusters, introduces many virtual edges to represent ordering. Re-calculation of the bottom-levels therefore takes up a lot of time.

The current implementation completely re-orders a cluster C_m after merging two clusters (C_1 and C_2). A possible optimization would be to improve the bottom-level calculation such that it can skip parts of the graph that did not change after merging. Furthermore the merging process itself can be improved by keeping the virtual edges in C_1 and C_2, and remap (and add were needed) virtual edges instead of merging the clusters, removing all virtual edges, and re-ordering the new cluster C_m again, which is the current implemented strategy.

Since *Internalization using Sarkar PA* is unable to finish with a minute on XEONx3, it fails to comply with one of the performance requirements.



(a) Average Calculate task timing diagram.

(b) Average Prepare task timing diagram.

Figure 6.5: Task execution timing diagrams for PPCx3 with *Internalization using Sarkar PA*

See Figure 6.1 for a description of all attributes.

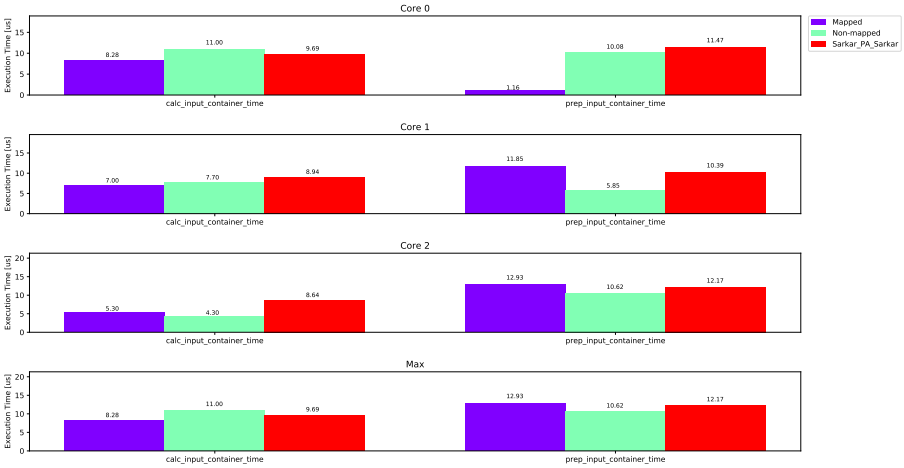
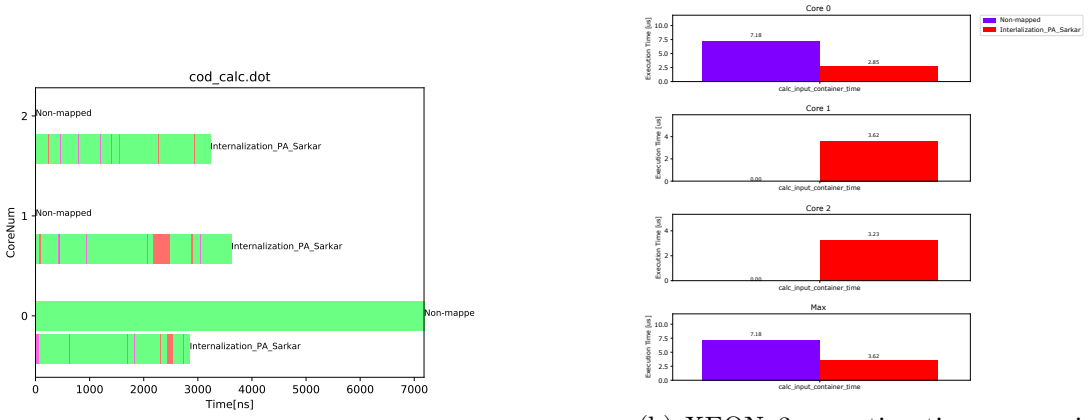


Figure 6.6: PPCx3 execution time comparison with *Internalization using Sarkar PA*.

See Figure 6.2 for a description of all attributes.



(a) Average Calculate task timing diagram.

(b) XEONx3 execution time comparison with *Internalization using Sarkar PA*.

Figure 6.7: XEONx3 results

See Figures 6.1 and 6.2 for the colour descriptions.

6.4.2 Load Balancing Processor Assignment

In the previous section, the *Internalization using Sarkar PA* algorithm was described. It was shown that the algorithm introduces long idle periods in the calculate schedule of PPCx3. It was shown that a potential cause could be the inaccuracy of the node measurements. It could however also be the case that the clusters are poorly chosen by the *Internalization* algorithm or the *Sarkar Processor Assignment* algorithm does not work well in general for PPCx3. In order to clarify this, a second PA algorithm was implemented that uses a more simplistic cluster merging tactic, which in addition, is

also expected to reduce the scheduling run-time. The *Load Balancing PA* algorithm uses Round Robin / Simple Load Balancing to equally divide the clusters created by the *Internalization* clustering algorithm, over the available hardware resources.

Surprisingly, despite having a relatively simplistic design, *Internalization using Load Balancing PA* performs $\sim 10.25\%$ better than *HLFET*, $\sim 6.2\%$ better than *Internalization using Sarkar PA*, $\sim 3.3\%$ better than the *Mapped* variant and $\sim 5.2\%$ better than the *Non-mapped* variant for PPCx3.

In XEONx3 *Internalization using Load Balancing PA* performs $\sim 12.3\%$ better compared to *HLFET* and $\sim 14.4\%$ worse than *Internalization using Sarkar PA*. The latter shows that schedulers may perform better or worse depending on the graph structure.

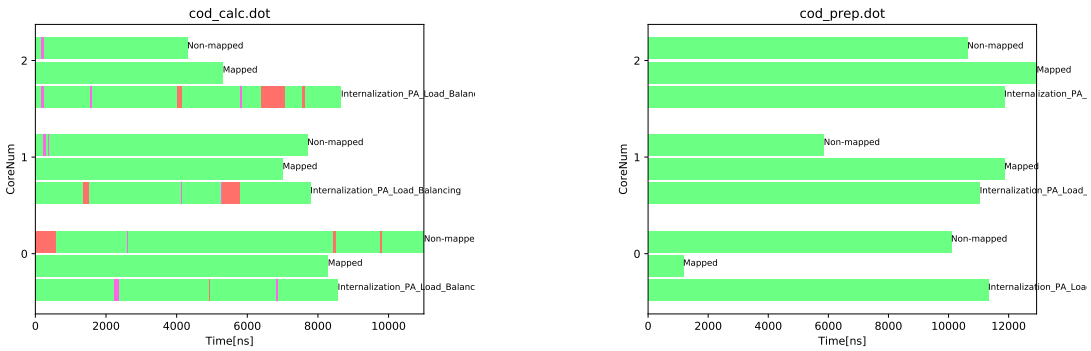
Table 6.5: *Internalization using Load Balancing PA* relative performance for PPCx3.

	Non-mapped	*Mapped*	*HLFET*	*Internalization using Sarkar PA*
Internalization using Load Balancing PA	5.19%	3.35%	10.25%	6.23%

Table 6.6: *Internalization using Load Balancing PA* relative performance for XEONx3.

	Non-mapped	*HLFET*	*Internalization using Sarkar PA*
Internalization using Load Balancing PA	42.38%	12.31%	−14.34%

On average *Internalization using Load Balancing PA* takes less than a second to schedule and assign both task graphs on PPCx3. On XEONx3, the algorithm requires 5 seconds to produce a valid schedule, which is far less than the *Sarkar PA* algorithm. *Internalization using Load Balancing PA* is the first algorithm, able to produce schedules within the 1 minute mark for all products, whilst meeting the performance requirements.



(a) Average Calculate task timing diagram.

(b) Average Prepare task timing diagram.

Figure 6.8: Task execution timing diagrams for PPCx3 with *Internalization using Load Balancing PA*.

See Figure 6.1 for a description of all attributes.

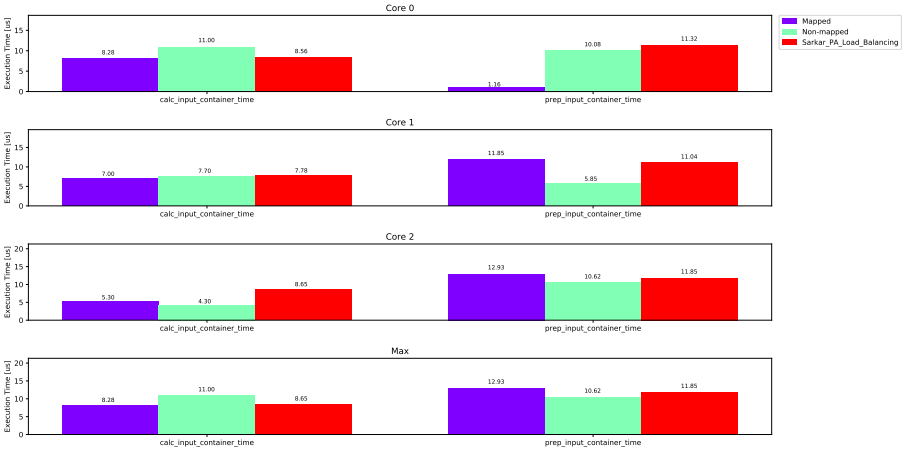
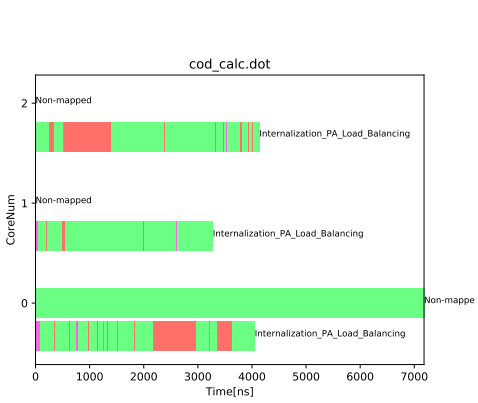
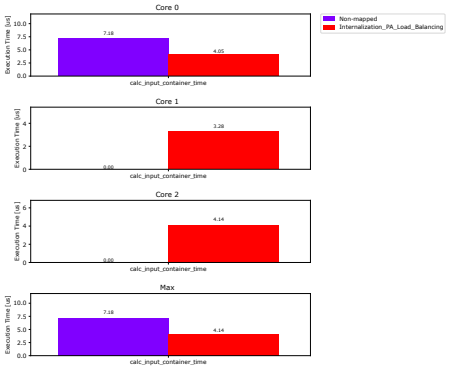


Figure 6.9: PPCx3 execution time comparison with *Internalization using Load Balancing PA*.

See Figure 6.2 for a description of all attributes.



(a) Average Calculate task timing diagram.



(b) XEONx3 execution time comparison with *Internalization using Load Balancing PA*.

Figure 6.10: XEONx3 results.

See Figures 6.1 and 6.2 for the color descriptions.

6.5 Insertion Scheduling Heuristic

ISH is based on the same metric as *HLFET*, that is, the Earliest Start Time. The *ISH* algorithm uses the insertion approach to fill in idle gaps of the schedule. Since *ISH* and *HLFET* share the same metric, parts of the *HLFET* implementation have been reused. The *ISH* algorithm has been implemented according to the steps described in Section 3.2.2.2. In Figure 6.11 and 6.12 the performance graphs of *ISH* are presented for PPCx3. In contrast to *HLFET*, *ISH* introduces far less core-to-core dependencies, because it takes into account edge weights. Furthermore it is shown that the idle times introduced

within the schedule have a relatively short duration compared to the results found in *Internalization using Sarkar PA* for example. In the schedule produced by *ISH*, each producer proceeds the corresponding consumer, which is desired when guards are inserted. Performance wise, the *ISH* algorithm is comparable with *Internalization using Sarkar PA* in PPCx3. The calculate phase is marginally faster than *Sarkar*, and the inverse holds for the prepare phase, leaving a negligible net difference. For XEONx3 however, the story is different. Within XEONx3, the performance of *ISH* is comparable with *Internalization using Load Balancing PA*, however, *Internalization using Sarkar PA* outperforms both by a significant amount. Compared to *HLFET*, *ISH* definitely introduces less idle periods within the schedule of XEONx3, however, *Internalization using Sarkar PA* introduces even less.

ISH performs $\sim 4.1\%$ better than *HLFET*, equal compared to *Internalization using Sarkar PA*, $\sim 6.9\%$ worse than *Internalization using Load Balancing PA*, $\sim 3.3\%$ worse than the *Mapped* variant and $\sim 1.3\%$ worse than the *Non-mapped* variant for PPCx3.

On XEONx3, *ISH* performs $\sim 11.8\%$ better than *HLFET*, $\sim 15.0\%$ worse than *Internalization using Sarkar PA* and is evenly matched with *Internalization using Load Balancing PA*.

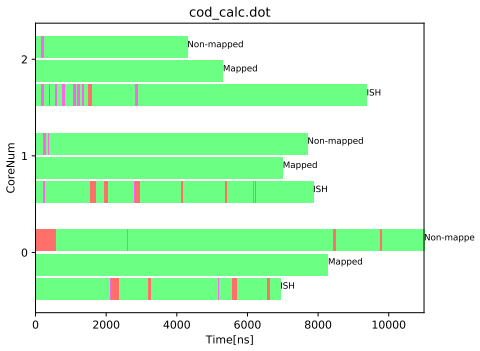
Table 6.7: *ISH* relative performance for PPCx3.

	Non-mapped	*Mapped*	*HLFET*	*Internalization using Sarkar PA*	*Internalization using Load Balancing PA*
ISH	−1.32%	−3.29%	4.09%	−0.21%	−6.87%

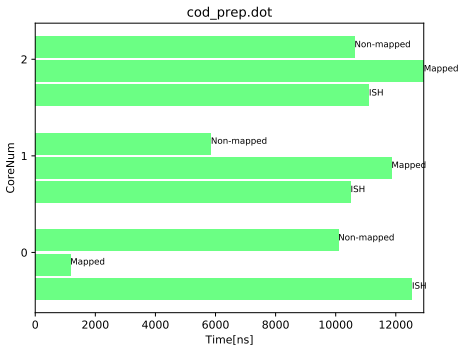
Table 6.8: *ISH* relative performance for XEONx3.

	Non-mapped	*HLFET*	*Internalization using Sarkar PA*	*Internalization using Load Balancing PA*
ISH	42.04%	11.79%	−15.01%	−0.59%

ISH requires less than a second to schedule both graphs on PPCx3. On XEONx3 however, *ISH* requires 475 seconds to produce a valid schedule, which is far beyond the one minute limit. As will be explained in the upcoming scalability analysis, the EST based algorithms do not seem to scale up very well. The implementation could probably be optimized (especially the EST calculation, which showed to be the main bottleneck during timing analysis), however, since both *HLFET* and *ISH* are overshadowed by the clustering based algorithms, there has been no effort put into optimizing the EST based schedulers.



(a) Average Calculate task execution timing diagram.



(b) Average Prepare task execution timing diagram.

Figure 6.11: CLC task execution timing diagrams for PPCx3 using *ISH*.

See Figure 6.1 for a description of all attributes.

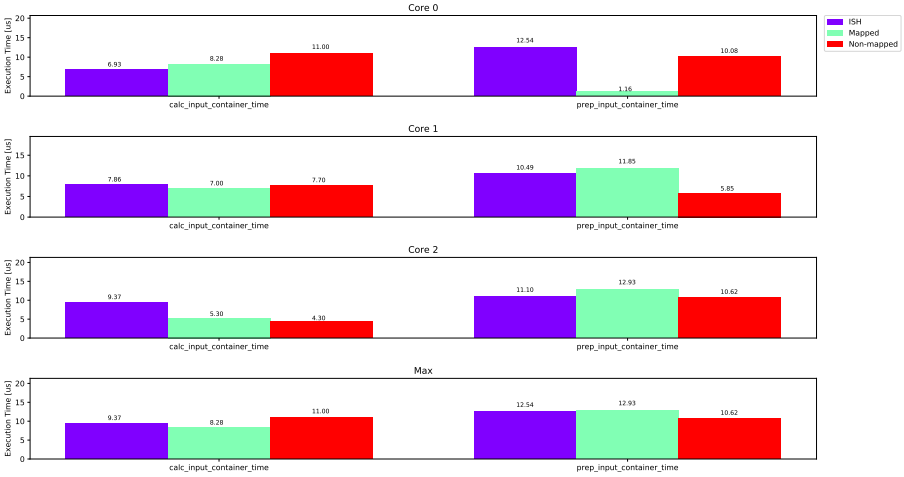
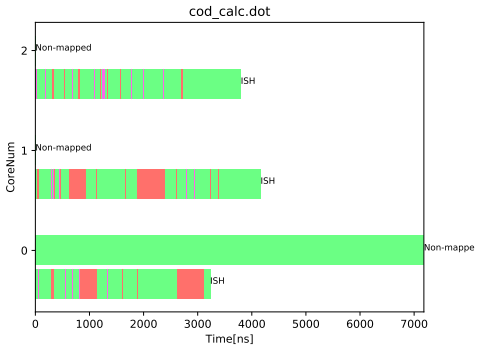
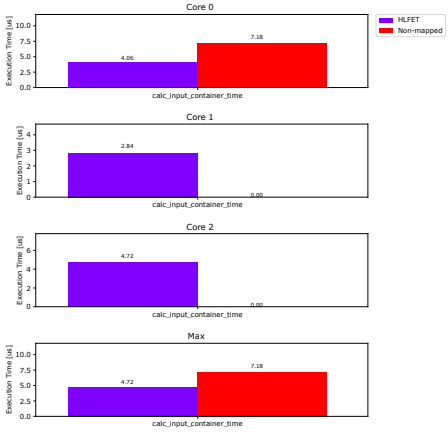


Figure 6.12: PPCx3 execution time comparison using *ISH*.

See Figure 6.2 for a description of all attributes.



(a) Average Calculate task timing diagram.



(b) XEONx3 execution time comparison using *ISH*.

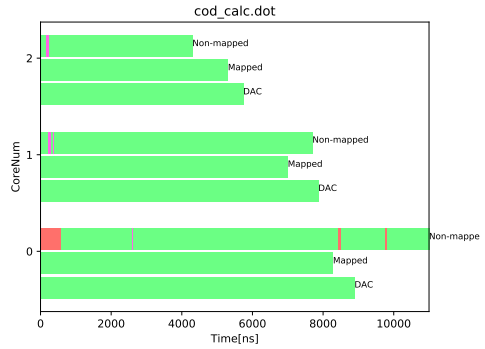
Figure 6.13: XEONx3 results.

See Figures 6.1 and 6.2 for the color descriptions.

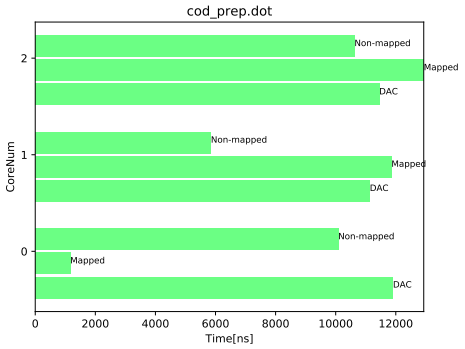
6.6 Dynamic Cluster Splitting

The last implemented scheduling algorithm, is the *DCS* method from Section 5.3.2. Each of the described splitting methods have been implemented, however, not all methods showed promising initial results. Furthermore, due to the limited amount of development time, a proper analysis of the methods was not possible. Not all implementations could be verified in time, furthermore no further improvements were possible. Therefore only the method will be discussed which showed the most promising performance results, in which the results can be used for future developments of the scheduling framework. The split strategy showing the most promising results was the the *Super Node* "Exposure" strategy in combination with *Internalization using Load Balancing PA* as internal scheduler.

Figure 6.14 and 6.15 show the performance of the *DCS* scheduler for PPCx3. It is shown that the *DCS* scheduler does not introduce any core to core dependency. According to the *DCS* scheduler, all subgraphs found within PPCx3 map quite well over the available cores, without requiring lots of splitting. The total run-time of *DCS* within PPCx3 is therefore relatively low, whilst it is still able to perform well.



(a) Average Calculate task timing diagram.



(b) Average Prepare task timing diagram.

Figure 6.14: Task execution timing diagrams for PPCx3 using *DCS*.

See Figure 6.1 for a description of all attributes.

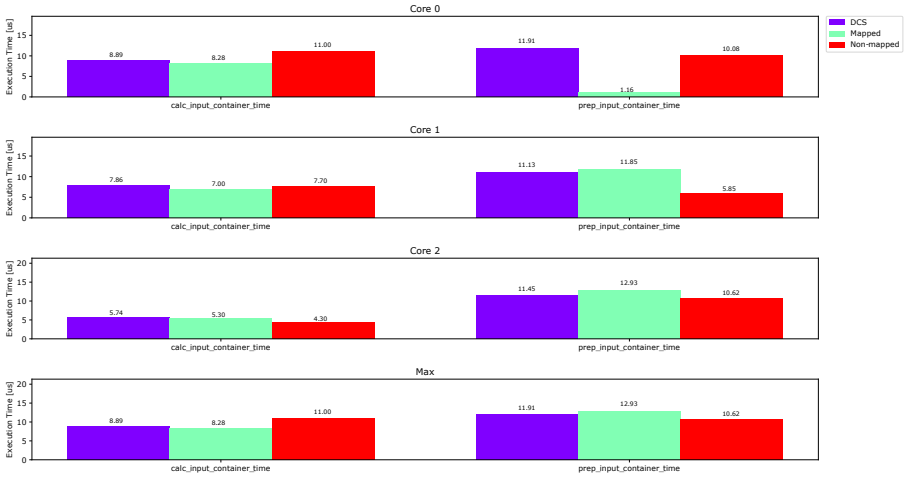


Figure 6.15: PPCx3 execution time comparison using *DCS*.

See Figure 6.2 for a description of all attributes.

In contrast to PPCx3, XEONx3 does not have many disjoint sub-graphs within its complete Directed Acyclic Graph. Therefore the use of the *DCS* scheduler is not beneficial using the super node exposure method, since it will expose all super nodes immediately, therefore constructing the initial graph again. Because of this, the internal scheduler (*Sarkar with Load Balancing PA* in this case) determines the complete scheduling process, thus showing no difference to the performance achieved using *Sarkar with Load Balancing PA* as stand-alone scheduling method.

All in all, *DCS* performs $\sim 9.0\%$ better than *HLFET*, $\sim 4.9\%$ better than *Internalization using Sarkar PA*, $\sim 1.4\%$ worse than *Internalization using Load Balancing PA*, $\sim 2.0\%$ better than the *Mapped* variant and $\sim 3.8\%$ better than the *Non-mapped* variant for PPCx3. Given that the performance of XEONx3 is equal to the performance of the internal scheduler, *DCS* is -like *Internalization using Load Balancing PA*- able to comply with all requirements. The total run-time for both products remains beneath a minute however scalability might be an issue, which is explained in Section 6.8.3.

Table 6.9: *DCS* relative performance for PPCx3.

	Non-mapped	*Mapped*	*HLFET*	*Internalization using Sarkar PA*	*Internalization using Load Balancing PA*	*ISH*
DCS	3.83%	1.96%	8.97%	4.89%	−1.43%	5.09%

6.7 Comparison

Figure 6.16 and 6.17 present the performance results for PPCx3 and Figure 6.18 presents the performance results for XEONx3. Combining both phases, *Internalization using Load Balancing PA* performs best within PPCx3, followed by the *DCS* scheduler. Within XEONx3, *Internalization using Sarkar PA* outperforms all other implementations. *DCS*

is left out of the graph in Figure 6.18 since XEONx3 does not contain many disjoint sub-graphs. Because of this, the node exposure technique, which is currently used, immediately exposes all nodes in the graph, meaning that the performance is equal to the internal scheduler that is used. In Table 6.10 and 6.11 the performance increase in percentage is presented for each scheduler compared to each other.

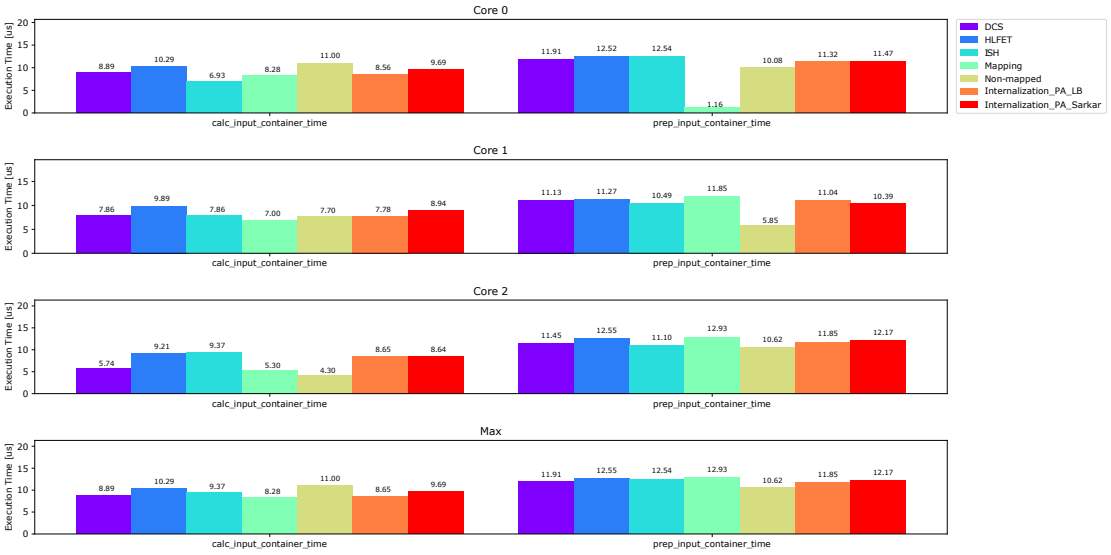


Figure 6.16: PPCx3 execution time comparison per phase.

Table 6.10: Performance Increase in Percentage for PPCx3.

Scheduler:	Non-mapped	Mapped	HLFET	Internalization PA Sarkar	Internalization PA LB	ISH	DAC
Non-mapped	0.00%	1.91%	-5.64%	-1.11%	5.19%	-1.32%	3.83%
Mapped		0.00%	-7.70%	-3.08%	3.35%	-3.29%	1.96%
HLFET			0.00%	4.29%	10.25%	4.09%	8.97%
Internalization PA Sarkar				0.00%	6.23%	-0.21%	4.89%
Internalization PA LB					0.00%	-6.87%	-1.43%
ISH						0.00%	5.09%
DAC							0.00%

Table 6.11: Performance Increase in Percentage for XEONx3.

Scheduler:	Non-mapped	HLFET	Internalization PA Sarkar	Internalization PA LB	ISH
Non-mapped	0.00%	34.30%	49.61%	42.38%	42.04%
HLFET		0.00%	23.31%	12.31%	11.79%
Internalization PA Sarkar			0.00%	-14.34%	-15.01%
Internalization PA LB				0.00%	-0.59%
ISH					0.00%

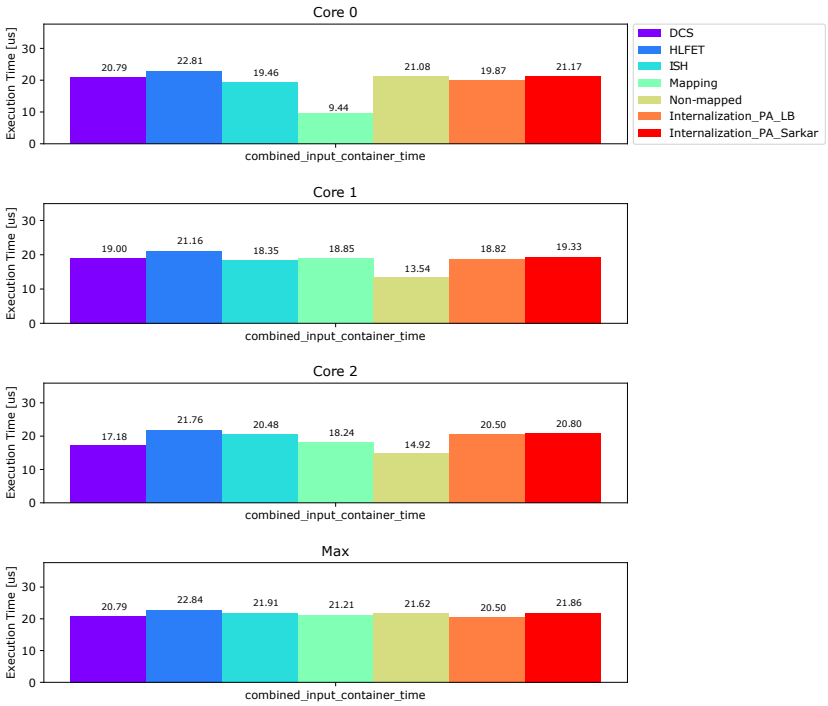


Figure 6.17: PPCx3 execution time comparison combined.

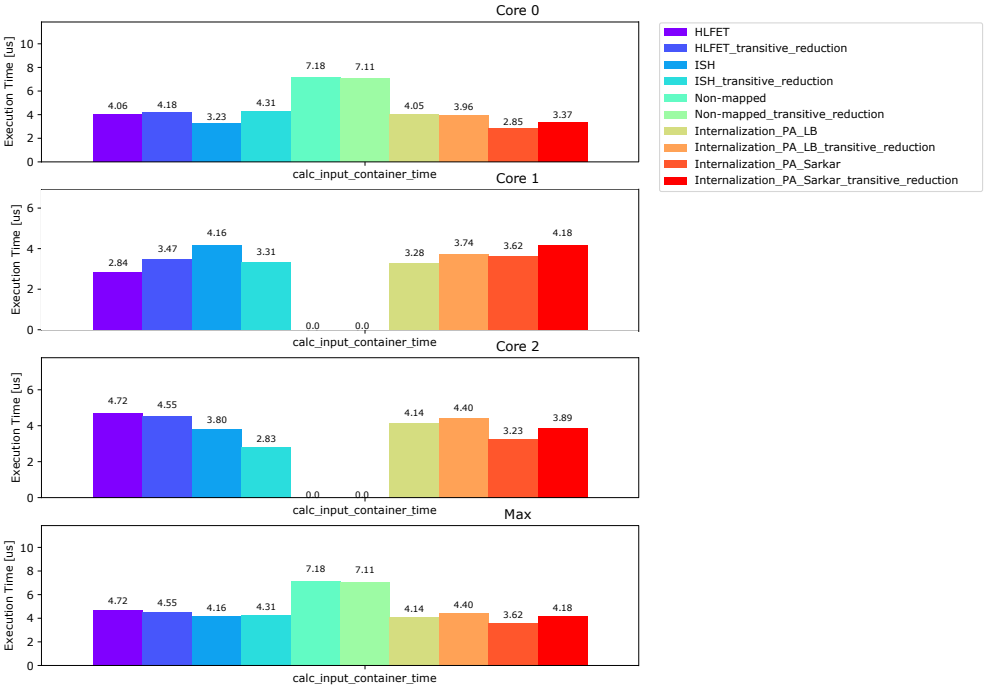


Figure 6.18: XEONx3 execution time comparison.

6.7.1 Transitive Reduction Results

Transitive reduction is only applicable in the XEONx3 product. Within XEONx3, 19 redundant edges could be removed from the graph. Using transitive reduction, as expected, the average scheduler run-time reduces for all implementations. However transitive reduction itself also costs time, making the net gain negligible within XEONx3. In Figure 6.18 the performance differences are presented for each scheduling implementation, with, and without transitive reduction. Unfortunately, in almost every case, transitive reduction worsens the schedule performance. Only *HLFET* seems to improve slightly, though the difference is too small in order to draw any conclusions. In Section 5.3.3 it was already mentioned that the performance might decrease, since, even though these dependencies are redundant, there is still a penalty to be paid when nodes connected through this redundant dependency are scheduled onto different cores.

6.8 Scalability Analysis

Even though most scheduler implementations are able to produce schedules for each product within a minute, it is important to analyze the scalability of each algorithm, because future products might contain more complex graphs. To check scalability, random graphs with varying sizes and edge probabilities were generated. To produce a random DAG, the steps presented in Appendix B.1 are followed. The graphs created using this method do not contain any cycles, however, these graphs may contain many redundant dependencies. To remove these, transitive reduction was applied.

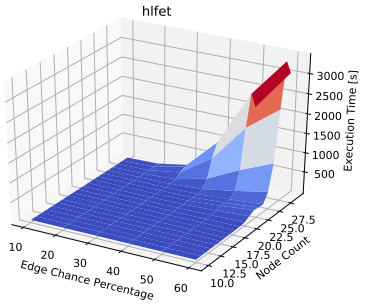
Note that increasing the amount of nodes for a randomly generated graph using a constant edge probability, also increases the amount of edges in the graph. The edge probability metric can therefore be interpreted as a unit-less quantity that indicates the level of complexity for each graph. Furthermore note that around 0.5 to 0.6, the most complex graphs are generated due to the transitive reduction step that is applied. When the edge probability exceeds this range, the graphs decrease in complexity because most edges are redundant and therefore terminated.

Comparing the randomly generated graphs to the graphs in PPCx3 and XEONx3, using probability ranges $[0.05, 0.10]$ and $[0.10, 0.20]$ respectively, the random graph generator produces the most similar graphs when comparing the amount of nodes per edge.

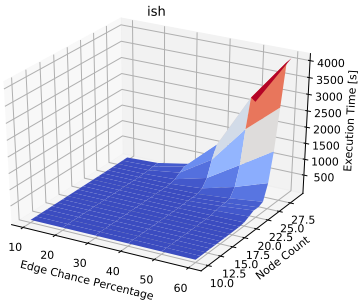
6.8.1 Varying Graph Size and Edge Probability

Figure 6.19 depicts the scalability results for graph sizes between 1 and 30 for *ISH* and *HLFET*, and 1 and 50 for the clustering algorithms. The scheduler run-time seem to increase exponentially with every scheduler, though for *ISH* and *HLFET*, the rate of growth is significantly larger compared to the clustering algorithms, which makes analysis difficult for graphs sizes exceeding 30 nodes, in combination with 0.6 edge probability. In the figures it is shown that the graphs for *ISH* and *HLFET* are similar, which is expected, since both algorithms use the same metric and base implementation. For *Internalization using Sarkar PA* and *Internalization using Load Balancing PA* a similar conclusion can be drawn. Both graphs appear to be identical, though *Internalization*

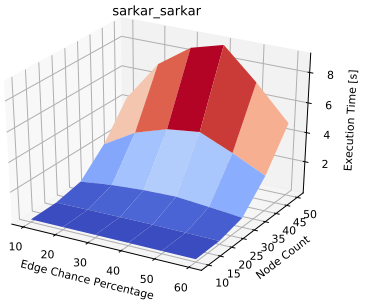
using Load Balancing PA has a lower overall run-time, which was also observed earlier during the implementation phase. Though the difference in run-time between the two clustering algorithms, observed within XEONx3 during the implementation phase, was much more significant than is presented in Figure 6.19d, in order to clarify this, a second analysis was performed.



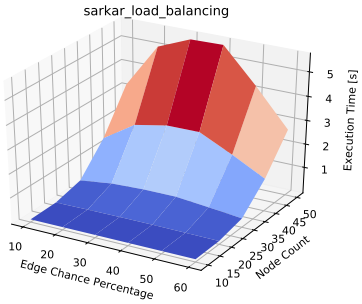
(a) HLFET



(b) ISH



(c) Internalization using Sarkar PA



(d) Internalization using Load Balancing PA

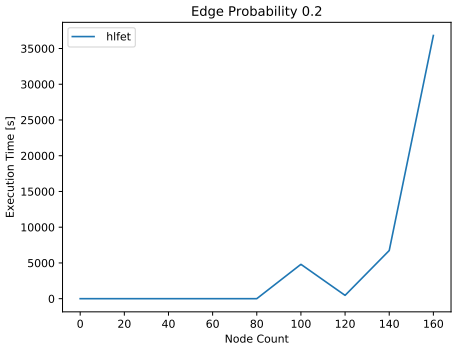
Figure 6.19: Scalability analysis for increasing edge probabilities and graph sizes.

The scalability analysis has been performed on the slowest multi-core platform: PPCx3.

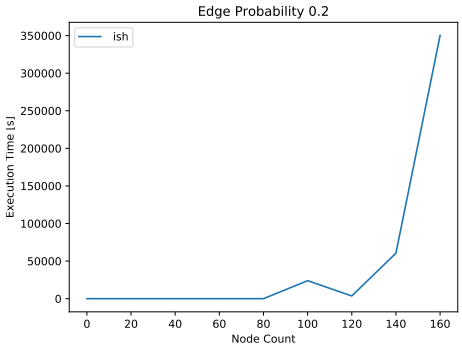
6.8.2 Constant Edge Probability

Given that the most complex graph in PMP (belonging to XEONx3), is at most as difficult as the randomly generated graphs using an edge probability of 0.2, a second analysis was performed using a constant edge probability of 0.2, though with a larger amount of nodes. The results are presented in Figure 6.20 and 6.21. Observing the results in Figure 6.19 it seemed as if *Internalization using Sarkar PA* (in contrary to the observations in the previous section) scales up at a similar rate as *Internalization using Load Balancing PA*. Furthermore *Internalization using Sarkar PA* seemed to scale up better than *HLFET*. However, using larger input graphs, it is shown that *HLFET* actually scales better than *Internalization using Sarkar PA*. Last but not least, the

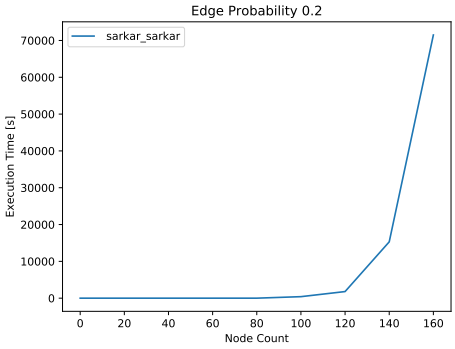
run-time of *Internalization using Load Balancing PA* seems to increase at a significantly lower rate compared to all the other implementations.



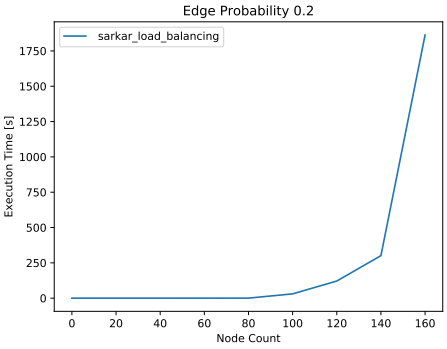
(a) HLFET



(b) ISH



(c) Internalization using Sarkar PA



(d) Internalization using Load Balancing PA

Figure 6.20: Scalability analysis for increasing graph sizes and constant edge probability of 0.2.

The scalability analysis has been performed on the slowest multi-core platform: PPCx3.

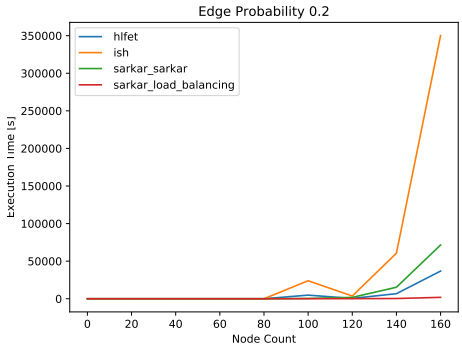


Figure 6.21: Combined scalability analysis for increasing graph sizes and constant edge probability of 0.2.

The scalability analysis has been performed on the slowest multi-core platform: PPCx3.

6.8.3 *DCS* Scalability

The scalability of *DCS* is mainly determined by the internal scheduler. Furthermore the scheduler run-time strongly depends on the graph. If the initial *Super Graph* maps well onto the available resources, the scheduling process is relatively short compared to all other scheduling implementations. However if the Parallel Time keeps decreasing with node splitting, the total run-time can grow significantly, which shows that the *DCS* scheduler in its current form should be improved, in order for it to be a reliable solution that can also be applied to any future products within PMP. Since the iterative approach cannot be used with a "dummy graph", the scalability analysis (as performed on all other methods) cannot be applied.

6.9 Conclusion

In this chapter an overview was given of all intermediate results that have been used to steer the iterative design and implementation approach. In addition, each scheduler implementation was compared against each-other, both in terms of resulting parallel time and overall scalability. It was shown that two of the implementations namely, *Internalization using Load Balancing PA* and *DCS*, in combination with Dynamic Measurements, were able to produce automated schedules whilst meeting all requirements. In the next chapter the conclusions are presented, as well as a discussion, and a future work presentation.

7 Conclusions

In this thesis an automated solution was proposed as alternative to the manual scheduling and assignation procedure within the Prodrive Motion Platform (PMP). The platform consists of generic mechatronic motion control software, that can be used in combination with a wide variety of products; ranging from elevators, to wafer-scanners. Due to the wide variety in products, there is also a wide variety in customer requirements. Therefore the hardware whereupon PMP operates, typically varies per product. Besides a wide variety in controllable hardware (drives, sensors, and actuators), the computing hardware, responsible for scheduling and assigning a customer's application, also varies on a product-to-product basis. Prior to this work, the scheduling and assignation of a customer's application to the available multi-core hardware, was based on time-consuming, manual procedures which cannot be efficiently utilized for future PMP products. In Chapter 2 and 4, the consequences of this problem were revealed by analyzing a recent PMP product called XEONx3. XEONx3 is orientated towards a model-based design flowed, which means that a customer is able to created their own control algorithm implementations, with unknown computational complexities, resulting in unpredictable task workloads. Given the fact that the original solution is based on a non-generic, time-consuming, manual scheduling flow with a limited amount of optimization possibilities, new applications like XEONx3, could not be scheduled directly.

In order to solve the problem at hand, a design for an automated framework was presented. The design consisted of two main parts, a scheduling and assignation framework, and a dynamic performance measuring framework. The measuring framework is used to discover the computational task load and estimate the penalty during core-to-core communication between two dependent tasks. Besides the dynamic measuring framework, a total of five different automated scheduling and assignation solutions were implemented, evaluated, and compared against each other in terms of schedule outcome and execution time.

In combination with the dynamic measurement framework, it was shown that two of the scheduling and assignation solutions, namely, *Internalization using Load Balancing PA* and *Dynamic Cluster Splitting*, are able to find schedules in an automated way, whilst still meeting the timing-constraints within current PMP applications. In contrast to the original solution, the new scheduling and assignation solution(s) allows to schedule and assign graphs with unknown structures and computational complexities, which, given the generic nature of PMP and the increasing demand for computational throughput, is a major advantage.

When comparing the average application execution time for typical control applications within existing PMP products, *Internalization using Load Balancing PA* provides performance improvements of $\sim 3.3\%$ and $\sim 5.2\%$ when compared to the *mapped* and *non-mapped* original scheduling solutions in , respectively. In terms of overall perfor-

mance, *Internalization using Load Balancing PA* is followed by *DCS*, which showed performance improvements of $\sim 2.0\%$ and $\sim 3.8\%$, respectively. Last but not least, it is shown that for XEONx3, in which both the *mapped* and *non-mapped* approaches resulted into a single core schedule, a performance improvement of 34.30% up to 49.61% can be observed, depending on the scheduling and assignation algorithm that is used. As additional requirement, the average schedule run-time should be kept below a minute for pre-existing products. Both *Internalization using Load Balancing PA* and *DCS* were able to meet this requirement for the considered applications.

Even though most scheduler implementations were able to produce schedules for each product within the one minute run-time requirement, a scalability analysis was performed with the future of PMP in mind. It was shown that *Internalization using Load Balancing PA* scales up relatively well compared to *ISH*, *HLFET* and *Internalization using Sarkar PA*. The scalability of the *DCS* algorithm depends on the internal scheduler and could not be analyzed in a straightforward way. Though in the worst case, *DCS* (in its current form) requires M iterations, in which M denotes the amount of disjoint sub-graphs in the initial graph. This means that, in the worst case, the internal scheduler is executed M times, in which the graph complexity increases at each iteration.

7.1 Future Work

In the end two automated scheduling methods were found which solve the main problem whilst meeting the performance criteria. However, as in most cases, the solution(s) can always be improved upon.

Static Edge Weight Measurements It was shown in Chapter 6 that the dynamic edge weight measurements showed some unexpected outliers. Moreover, the measured execution time of a tasks during the dynamic node weight measurements, may not always correspond with the resulting execution time within a specific schedule. As mentioned in Chapter 6, the latter is difficult to take into account, though the former might be solvable when the dynamic edge weight measurements are combined with static measurements. Given that the performance of the schedulers strongly depend on the accuracy of the measurements, an updated edge measurements routine may enable more scheduling implementations to meet the performance requirements, which may lead to different conclusions.

Increasing Timer Precision on PPCx3 As shown in the analysis phase, the precision of the system timer used within PPCx3 is quite low compared to the duration of some small tasks within the system (for example the sensor tasks). This means that differences in computational time between tasks that are smaller than the timer precision, will never be available to the scheduler. Increasing the precision of the measurements is expected to improve scheduling decisions, thus improving the performance. However the hardware whereupon PPCx3 is based does not support more accurate timers than was already available. Thus the only way to support higher precision timers within PPCx3, is to change the hardware.

Inclusion of all Tasks Within this research, only the sub-controller DAG (which constitutes the customer's application), is taken into account, since the platform in its current state, does not allow for dynamic (re)scheduling of non-customer related tasks. Even though this particular task set is the most complex to schedule, including the complete-task set would be beneficial for the overall system performance. For example the coarse grained synchronization points (see Section 2.4), could be represented by fine grained edge dependencies in order to broaden the parallelization possibilities.

Improving the *DCS* Scheduler The *DCS* scheduler showed promising initial results and can be used to schedule graphs containing many disjoint sub-graphs. Since disjoint sub-graph structures are likely to reoccur in future PMP products, it is worthwhile to seek design solutions able to alleviate its scalability issue. The main bottleneck is the internal scheduler, which reschedules the complete graph at every iteration. A possible improvement would be to perform a partial rescheduling, in which only the newly created *Super Node*s are fitted within the *Super Graph* schedule, determined in the previous iteration. Last but not least, the different node splitting techniques could be combined and optimized, to further improve the performance.

Statistical Scheduling Another interesting yet unimplemented design step is statistical measurements. As discussed in Chapter 4 and 5, the computational demand (or weight) of a task, is a multi-valued metric due to the conditional execution behaviour. With the use of statistical measurements, the robustness of the schedule and assignation, can be improved in order to make it more resilient against execution time fluctuations.

Bibliography

[1] J. D. Ullman, "Np-complete scheduling problems," *Journal of Computer and System sciences*, vol. 10, no. 3, pp. 384–393, 1975.

[2] M. Fujii, T. Kasami, and K. Ninomiya, "Optimal sequencing of two equivalent processors," *SIAM Journal on Applied Mathematics*, vol. 17, no. 4, pp. 784–789, 1969.

[3] E. G. Coffman and R. L. Graham, "Optimal scheduling for two-processor systems," *Acta informatica*, vol. 1, no. 3, pp. 200–213, 1972.

[4] M. R. Garey and D. S. Johnson, "Two-processor scheduling with start-times and deadlines," *SIAM Journal on Computing*, vol. 6, no. 3, pp. 416–426, 1977.

[5] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Commun. ACM*, vol. 17, no. 12, pp. 685–690, Dec. 1974. [Online]. Available: http://doi.acm.org/10.1145/361604.361619

[6] B. Kruatrachue, "Static task scheduling and grain packing in parallel processing systems," Ph.D. dissertation, Corvallis, OR, USA, 1987, aAI8806917.

[7] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Cambridge, MA, USA: MIT Press, 1989.

[8] S. J. Kim and J. C. Browne, "A general approach to mapping of parallel computations upon multiprocessor architectures," *Proceedings of the International Conference on Parallel Processing*, vol. 3, pp. 1–8, 12 1988.

[9] A. Gerasoulis and T. Yang, "A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors," *Journal of Parallel and Distributed Computing*, vol. 16, no. 4, pp. 276 – 291, 1992.

[10] T. Yang and A. Gerasoulis, "Dsc: Scheduling parallel tasks on an unbounded number of processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 9, pp. 951–967, 1994.

[11] M. Y. Wu and D. D. Gajski, "Hypertool: a programming aid for message-passing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 3, pp. 330–343, Jul 1990.

[12] R. P. Bianchini, Jr and J. P. Shen, "Interprocessor traffic scheduling algorithm for multiple-processor networks," *IEEE Trans. Comput.*, vol. 36, no. 4, pp. 396–409, Apr. 1987. [Online]. Available: http://dx.doi.org/10.1109/TC.1987.1676922

[13] J.-C. Liou and M. A. Palis, "A comparison of general approaches to multiprocessor scheduling," in *Parallel Processing Symposium, 1997. Proceedings., 11th International.* IEEE, 1997, pp. 152–156.

[14] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, no. 4, pp. 406–471, Dec. 1999. [Online]. Available: http://doi.acm.org/10.1145/344588.344618

[15] T. M. Mitchell, *Machine Learning*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997.

[16] A. Shahid, M. S. Benten, and S. M. Sait, "Gsa: Scheduling and allocation using genetic algorithm," in *Proceedings of the Conference on European Design Automation*, ser. EURO-DAC '94. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 84–89. [Online]. Available: http://dl.acm.org/citation.cfm?id=198174.198218

[17] V. A. Nguyen, D. Hardy, and I. Puaut, "Cache-conscious offline real-time task scheduling for multi-core processors," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Bertogna, Ed., vol. 76. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 14:1–14:22. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/7164

[18] A. Lindsay and B. Ravindran, "On cache-aware task partitioning for multicore embedded real-time systems," in *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS)*, IEEE. IEEE, 2014, pp. 677–684.

[19] N. Guan, M. Stigge, W. Yi, and G. Yu, "Cache-aware scheduling and analysis for multicores," in *Proceedings of the seventh ACM international conference on Embedded software*. ACM, 2009, pp. 245–254.

[20] P. Choudhury, P. P. Chakrabarti, and R. Kumar, "Online scheduling of dynamic task graphs with communication and contention for multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 1, pp. 126–133, Jan 2012.

[21] N. R. Satish, K. Ravindran, and K. Keutzer, "Scheduling task dependence graphs with variable task execution times onto heterogeneous multiprocessors," in *Proceedings of the 8th ACM international conference on Embedded software*. ACM, 2008, pp. 149–158.

[22] *e500mc Core Reference Manual*, Freescale Semiconductor.

Appendices

Experimental Data

A

A.1 PPCx3 execution time table for the *mapped* scheduler

Core	Order	TaskID	Type	Avg[ns]	Min[ns]	Max[ns]	Std	WCC
0	0	4	Sensor	347.181	256	682	31.359	384
0	1	3	ControlNetwork	1453.33	1237	3499	131.164	1365
0	2	7	ControlNetwork	826.803	768	1366	22.4699	811
0	3	1	Matrix	411.13	341	726	31.2309	426
0	4	11	Sensor	257.36	213	299	10.2178	256
0	5	9	Matrix	312.154	298	640	20.2219	342
0	6	8	Sensor	580.263	469	1408	54.6748	597
0	7	0	Actuator	694.845	597	1450	33.7815	640
0	8	14	ControlNetwork	883.025	853	1450	28.9396	853
0	9	13	Actuator	557.59	512	768	23.5535	512
0	10	16	Sensor	381.605	341	427	13.1189	384
0	11	15	Actuator	535.935	512	683	21.3923	555
0	12	19	Sensor	256.892	213	299	9.22784	256
0	13	18	ControlNetwork	820.712	768	1493	22.3449	811
0	14	17	Actuator	554.324	512	725	12.9487	554
0	15	28	Sensor	257.865	213	469	11.1736	256
0	16	27	ControlNetwork	855.511	810	1536	33.0015	811
0	17	26	Actuator	540.317	512	726	20.5957	512

Core	Order	TaskID	Type	Avg[ns]	Min[ns]	Max[ns]	Std	WCC
1	0	22	Sensor	224.989	170	512	26.7299	341
1	1	21	ControlNetwork	1237.14	1109	4821	117.66	4821
1	2	20	Actuator	276.142	170	598	29.5594	426
1	3	25	Sensor	168.986	128	256	9.04383	171
1	4	24	ControlNetwork	949.964	896	1750	33.4085	1152
1	5	23	Actuator	736.076	554	2176	58.5107	2176
1	6	34	Sensor	167.402	128	342	12.4979	256
1	7	33	ControlNetwork	896.871	853	2218	42.99	939
1	8	32	Actuator	480.4	426	726	22.1202	512
1	9	37	Sensor	164.528	128	299	15.0528	170
1	10	36	ControlNetwork	858.765	810	4096	46.3869	1024
1	11	35	Actuator	217.96	170	512	26.3614	213
1	12	40	Sensor	163.156	128	256	16.2947	128
1	13	39	ControlNetwork	860.59	810	1536	27.9345	1066
1	14	38	Actuator	191.606	170	342	22.2471	171
1	15	63	Sensor	170.341	128	298	7.45826	170
1	16	62	ControlNetwork	911.138	853	1578	24.9533	982
1	17	61	Actuator	472.209	426	598	14.1115	597
1	18	66	Sensor	163.452	128	469	16.3037	171
1	19	65	ControlNetwork	880.32	853	1536	25.3983	896
1	20	64	Actuator	465.336	426	640	15.6992	469

Core	Order	TaskID	Type	Avg[ns]	Min[ns]	Max[ns]	Std	WCC
2	0	31	Sensor	244.101	213	512	20.0022	256
2	1	30	ControlNetwork	1263.53	1109	4736	79.4866	1238
2	2	29	Actuator	824.522	725	2304	53.5687	810
2	3	43	Sensor	257.945	213	512	12.5133	256
2	4	42	ControlNetwork	870.028	810	2134	30.2153	939
2	5	41	Actuator	579.331	469	853	40.1579	554
2	6	48	Sensor	161.693	128	171	17.3879	171
2	7	50	Sensor	163.066	128	342	16.8838	171
2	8	46	Matrix	353.44	341	854	20.5769	384
2	9	45	Sensor	445.703	426	853	23.8738	427
2	10	55	ControlNetwork	874.259	810	1707	31.2159	853
2	11	56	ControlNetwork	862.369	810	1664	27.7712	810
2	12	53	Matrix	368.261	298	683	25.8796	384
2	13	44	Actuator	479.146	426	768	22.8482	469
2	14	58	Sensor	470.157	426	640	10.288	469
2	15	57	Actuator	466.311	426	768	16.9522	470
2	16	60	ControlNetwork	888.298	853	1750	24.1136	896
2	17	59	Actuator	500.929	469	683	18.8634	512

A.2 PPCx3 execution time table for the *non-mapped* scheduler

Core	Order	TaskID	Type	Avg[ns]	Min[ns]	Max[ns]	Std	WCC
0	0	2	Sensor	286.564	213	640	34.9248	299
0	1	1	ControlNetwork	1502.59	1280	3029	99.8816	1664
0	2	0	Actuator	911.929	725	1579	61.1002	938
0	3	20	Sensor	169.263	128	469	19.1412	171
0	4	19	ControlNetwork	831.399	768	1750	35.1384	896
0	5	18	Actuator	455.648	426	725	20.2704	469

Core	Order	TaskID	Type	Avg[ns]	Min[ns]	Max[ns]	Std	WCC
1	0	n/a	Consumer	1073.54	640	5931	201.064	1024
1	1	57	ControlNetwork	1222.52	1109	4864	109.347	1280
1	2	n/a	Producer	141.67	128	171	19.9093	171
1	3	8	Sensor	179.181	128	342	17.4017	171
1	4	7	ControlNetwork	871.087	810	1878	39.6825	896
1	5	6	Actuator	256.127	170	512	33.3748	256
1	6	14	Sensor	160.116	128	171	18.4103	170
1	7	13	ControlNetwork	837.964	768	1707	30.5065	854
1	8	12	Actuator	182.049	170	299	19.0121	170
1	9	31	Sensor	184.476	128	512	25.5417	171
1	10	34	Sensor	156.436	128	298	20.2056	171
1	11	29	Matrix	362.634	298	810	38.7846	341
1	12	28	Sensor	544.058	427	1536	38.9228	555
1	13	38	ControlNetwork	852.303	810	1536	26.1173	853
1	14	39	ControlNetwork	850.125	810	1664	28.4495	896
1	15	36	Matrix	392.502	298	683	38.6752	427
1	16	27	Actuator	605.744	469	1195	37.9984	640
1	17	41	Sensor	383.845	341	427	13.1362	384
1	18	40	Actuator	462.205	426	683	17.4572	469
1	19	43	ControlNetwork	849.718	810	1750	26.4372	853
1	20	42	Actuator	465.44	426	640	16.4179	469
1	21	n/a	Consumer	213.343	170	256	11.9592	214
1	22	66	ControlNetwork	899.763	853	1834	20.5863	896
1	23	65	Actuator	467.763	426	598	13.3875	469

Core	Order	TaskID	Type	Avg[ns]	Min[ns]	Max[ns]	Std	WCC
2	0	54	Sensor	238.379	170	725	25.2981	213
2	1	n/a	Producer	157.121	128	384	20.6195	171
2	2	61	Sensor	254.546	213	299	10.9048	256
2	3	n/a	Producer	142.67	128	171	20.2688	171
2	4	5	Sensor	172.332	128	427	9.74228	170
2	5	4	ControlNetwork	1310.98	1110	4822	121.106	1280
2	6	3	Actuator	670.35	554	2091	54.8781	598
2	7	11	Sensor	161.897	128	299	17.3787	170
2	8	10	ControlNetwork	903.683	853	2475	44.0695	939
2	9	9	Actuator	480.299	426	811	25.0763	469
2	10	17	Sensor	173.914	170	214	11.3138	171
2	11	16	ControlNetwork	882.121	810	1707	34.1939	896
2	12	15	Actuator	254.197	170	640	30.1317	256
2	13	23	Sensor	165.741	128	256	13.7813	171
2	14	22	ControlNetwork	868.908	810	1621	28.8463	896
2	15	21	Actuator	470.006	426	768	25.2293	469
2	16	26	Sensor	172.944	128	469	18.9136	128
2	17	25	ControlNetwork	862.72	810	1579	30.1434	853
2	18	24	Actuator	501.695	426	810	23.4956	512
2	19	46	Sensor	166.718	128	171	12.3653	171
2	20	45	ControlNetwork	1005.47	896	1835	30.9374	1024
2	21	44	Actuator	468.283	426	768	23.5718	469
2	22	49	Sensor	176.226	128	512	20.3326	171
2	23	48	ControlNetwork	957.521	896	1664	31.0409	981
2	24	47	Actuator	472.296	426	683	19.7172	512
2	25	53	ControlNetwork	905.158	853	1664	23.0596	896
2	26	n/a	Consumer	248.509	213	554	29.0218	299
2	27	51	Matrix	516.776	384	768	39.4178	512
2	28	59	Matrix	401.756	298	683	32.2309	426
2	29	58	Sensor	460.463	384	1024	30.8076	427
2	30	50	Actuator	476.471	426	768	19.1579	469
2	31	64	Sensor	389.62	341	512	15.0947	384
2	32	63	Actuator	569.418	512	768	23.2308	555

A.3 XEONx3 Assigned Calculate DAG

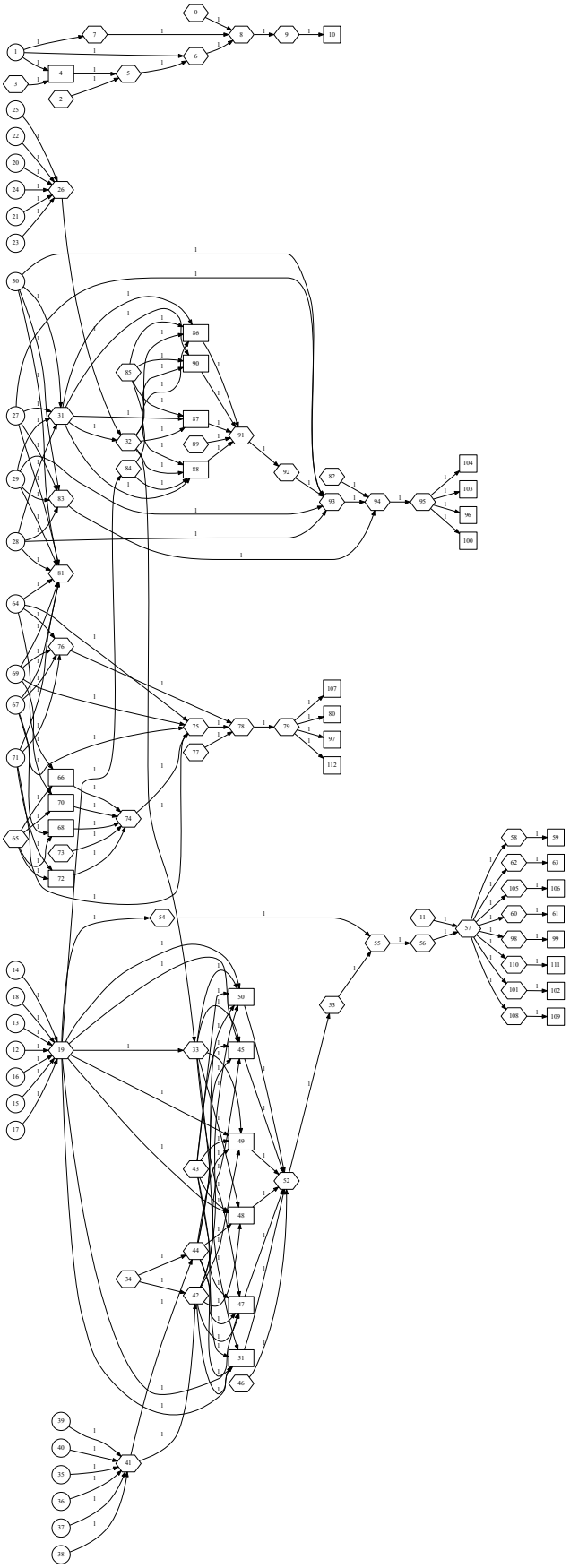


Figure A.1: Assigned calculate DAG for the *non-mapped* (and *mapped*) scheduler in XEONx3

See Figure 4.1 for a description of the colours.

A.4 Dynamic Measurements Results

In the following figures, the Dynamic Measurements results are shown for PPCx3 and XEONx3.

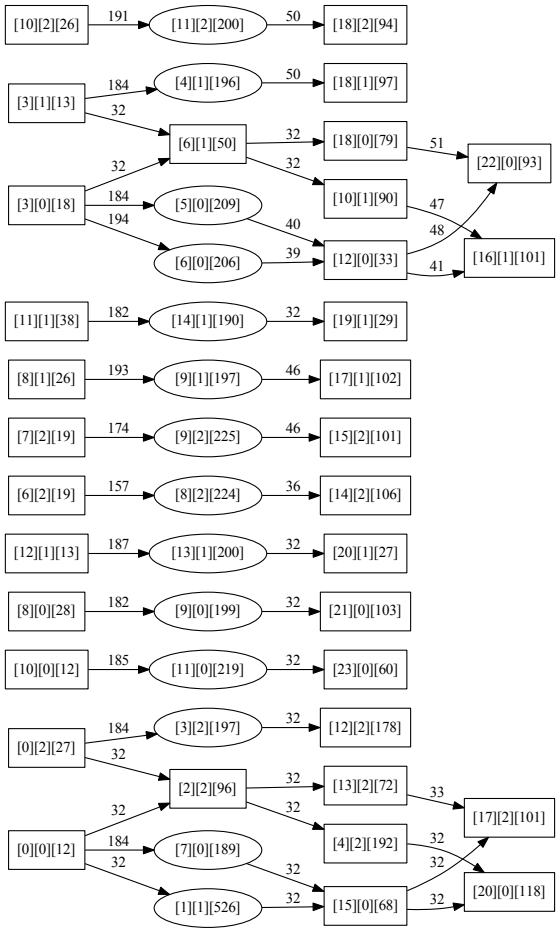


Figure A.2: PPCx3 Directed Acyclic Graph including node and edge weights found during dynamic measurements

The last [] entry in the node labels indicate the node weight, the edge labels represents the edge weights. The weights in the graph correspond to the average amount of execution cylces upscaled by a factor of 10. As already mentioned the precision of the PPCx3 timer is relatively low compared to the computational load of some of the tasks.

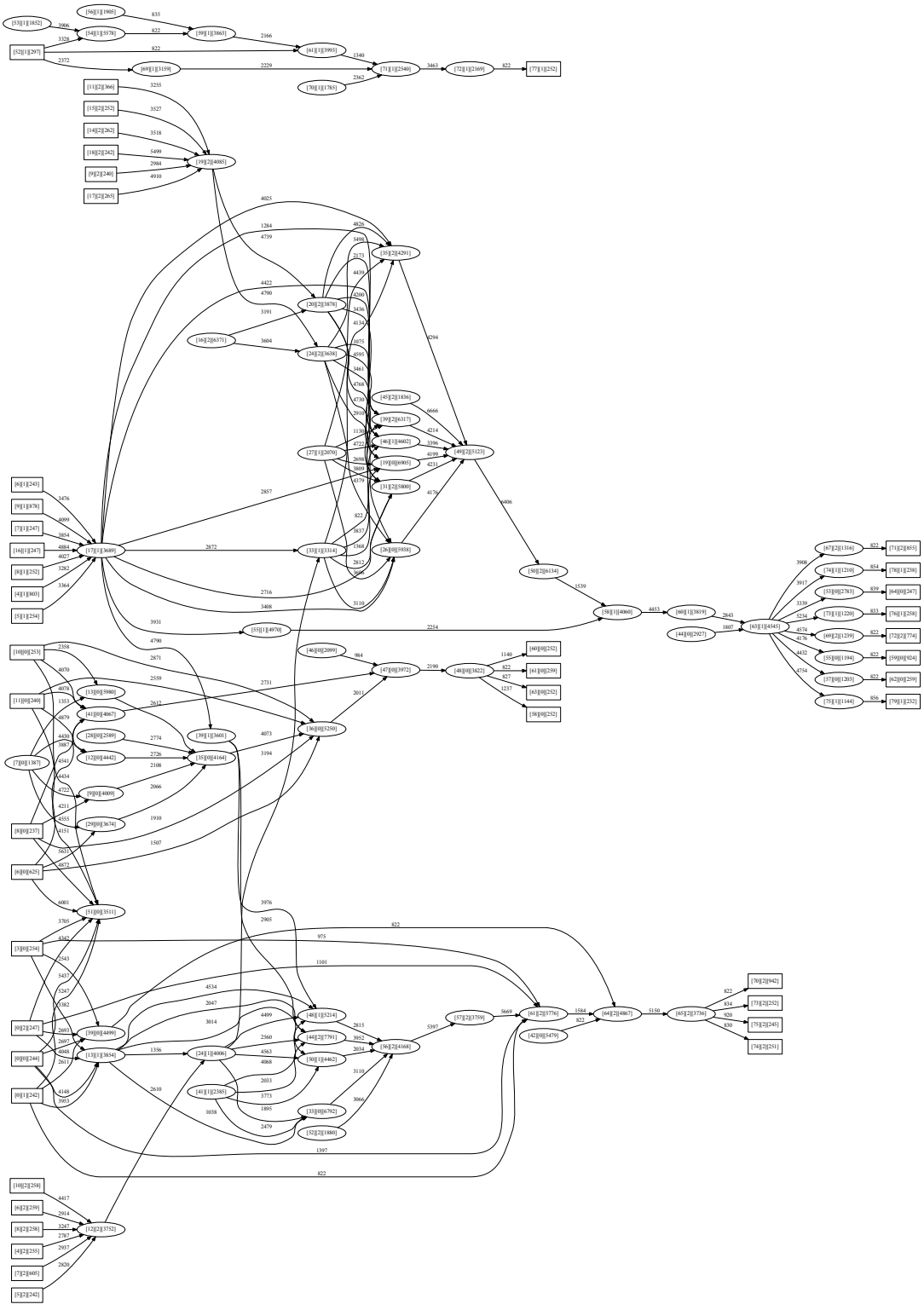
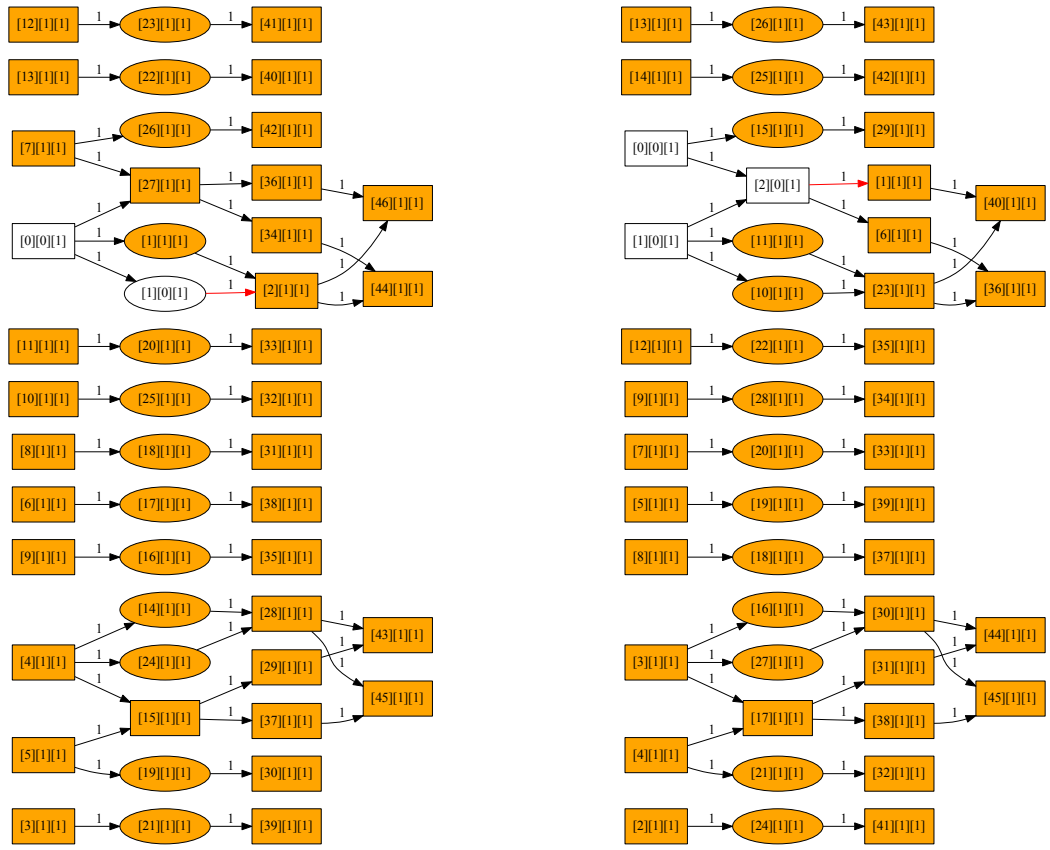


Figure A.3: XEONx3 Directed Acyclic Graph including node and edge weights found during dynamic measurements

The last [] entry in the node labels indicate the node weight, the edge labels represents the edge weights. The weights in the graph correspond to the average amount of execution cylces upscaled by a factor of 10.

A.5 Dynamic Edge Measurements Results

In the following figure, two example iterations of edge measurements in PPCx3 are shown.



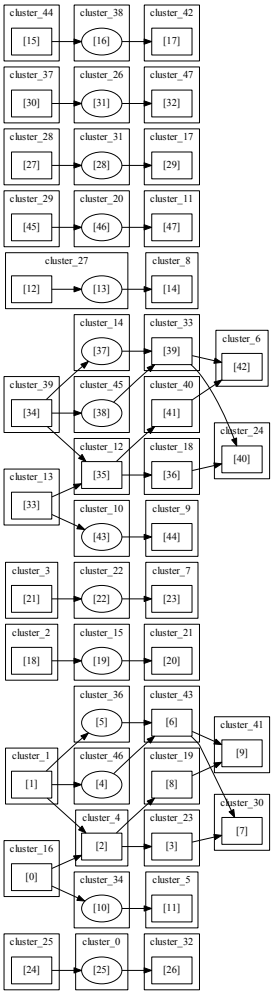
(a) Isolated edge measurement example 1

(b) Isolated edge measurement example 2

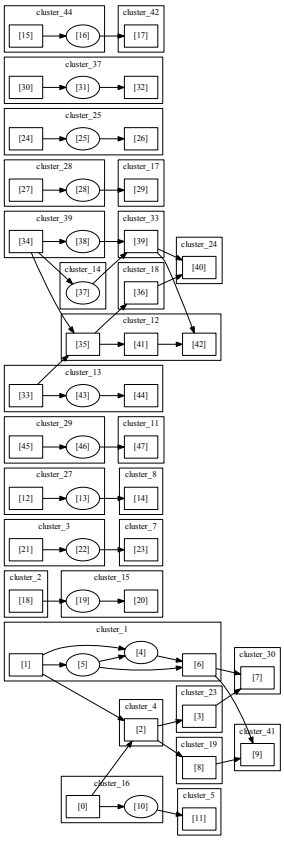
Figure A.4: Two isolated edge measurement iterations from PPCx3 used as example

The edge in red depicts the edge under measurement. The node labels are specified as [ordering number][core number][n/a], note that, since the producer and consumer tasks are not explicitly drawn, the [ordering number] of the destination core starts at [1] (since there is a consumer that proceeds execution), furthermore the [ordering number] of the source core skips a number which is assigned to the producer. The edge labels should be ignored.

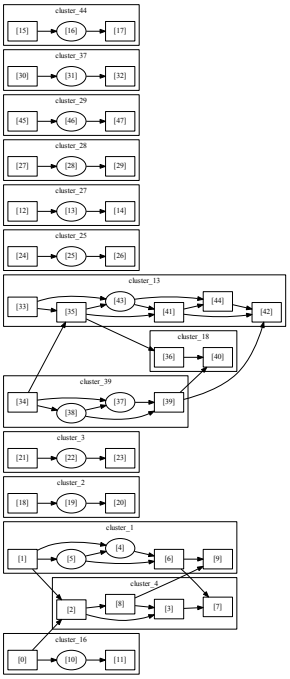
A.6 Internalization Results



(a) Iteration 0 (start)



(b) Iteration 20



(c) Iteration 41 (end)

Figure A.5: Some *Internalization* iterations extracted from PPCx3

Pseudo-code

B

B.1 Random Graph Creation Steps

A random graph is created using the following steps:

1. Input: edge chance percentage P_{edge} and the graph size / amount of nodes N.
2. Create a list of N "dummy" nodes L_n.
3. Create another list of nodes $L_{unprocessed}$ initialized with the same nodes as L_n.
4. For all nodes $n_x \in L_n$:
   (a) Remove n_x from the unprocessed list $L_{unprocessed}$.
   (b) Assign a random weight to n_x.
   (c) For all unprocessed nodes $n_y \in L_{unprocessed}$:
      i. Create an edge between n_x and n_y with edge creation chance P_{edge}.