MASTER

Load balancing of real-time tasks over the multicore multihost system

Patwardhan, S.A.

Award date:
2012

Link to publication
Load Balancing of Real-Time Tasks Over The Multicore Multihost System
Siddhesh Patwardhan: 0756928

Course code 2IM90

Computer Systems
ASML, Veldhoven

System Architecture and Networking
Department of Mathematics and Computer Science
Eindhoven University of Technology

University Supervisor Dr.ir. R. J. Bril
ASML Supervisors Gertjan Schouten and Erik Niessen
Abstract

ASML is one of the world’s leading providers of lithography systems for the semiconductor industry, manufacturing complex machines that are critical to the production of integrated circuits or microchips. These machines are in a phase of constant upgrading to achieve higher throughput. Also, accuracy of these machines is increasing as they work with smaller and smaller critical dimensions and overlay.

The lithography machines built by ASML consist of two parts, a main host part and an embedded part. The embedded part is responsible for functionality and interaction with sensors and actuators while the main host part performs controlling tasks. The focus of the thesis is on the main host part. The main host executes a combination of real-time and non-real time tasks on a single multicore host. Currently, the operating system’s scheduler decides the mapping of these tasks on the available cores. This mapping is dynamic rather than static and can create unnecessary resource sharing between the tasks. Hence this kind of mapping can not always guarantee that the required throughput can be achieved. This makes the machine’s throughput unpredictable.

Moreover, ASML is in phase of migrating the main host from a single multicore host to multiple multicore hosts. After the migration it is not possible for the operating system’s scheduler to find out the feasible schedule considering overall processing power of all the hosts. It means that the software is not scalable with the hardware. The operating system also does not know about the critical path in a given application. Randomly assigning the tasks to different hosts is not an option as it can have negative impact on the throughput because of the communication overhead.

This thesis deals with generating a feasible mapping of the tasks on the available hosts and their cores with the aim to improve the current throughput of the machine, making it predictable and making the software on the main host scalable with the hardware. As a first step, characteristics of the platform and tasks running on the main host are analyzed and a task graph showing the behaviour of the control application is created. As this task graph turned out to be mostly sequential, general task sets having parallel tasks are considered for mapping.

For generating a mapping for a generalized task set, a mapping algorithm is developed. Such a mapping algorithm should be able to find a feasible mapping over available multicore hosts such that the throughput of the machine is predictable. The developed mapping algorithm is tested by using POOSL model simulations as well as on real hardware. It was seen that the mapping created by the algorithm performs better than the schedule created by the operating system’s scheduler in terms of overall finish time of a task set or throughput.
Preface

This thesis reflects eight months of the thesis work done at ASML, Eindhoven to conclude the Embedded Systems Master study. This work is done within the computer systems group at ASML together with System Architecture and Networking (SAN) group at the Technical University of Eindhoven.

ASML is one of the world’s leading providers of lithography machines for the semiconductor industry. The computer systems department at ASML deals with operating systems, networking and resource monitoring and management of limited resources like hard disk space, memory present in these machines. It also deals with the tracing of execution threads for the Solaris operating system.

The main research fields of the SAN group is on the architecture of networked embedded systems, including hardware and software aspects.

The lithography machines built by ASML consists of two parts, the main host part and the embedded part. This thesis was initially about balancing the main host’s tasks across multiple hosts having multiple cores, in order to improve the throughput of the machine as well as make it deterministic and make software scalable with hardware upgrades. The first three months of the thesis were spent in assessing the feasibility of the assignment. The solution to the load balancing problem greatly depends on the characteristics of the tasks to be balanced as well as the platform characteristics. During the feasibility study focus was on understanding the system working, creating an application model and a resource model as well as exploring the tools to be used.

After the feasibility study, a task graph has been constructed which suggested that the main host tasks are mostly sequential and there is not much parallelism to exploit. But in the future, there will be a need to parallelize these tasks in order to achieve higher throughput. Hence the initial problem statement has been revised such that, general task sets having similar characteristics as those of the main host but having enough parallelism are considered.

The main aim of the thesis is given a platform model, try to find a feasible deployment, such that the critical path of the task set is satisfied. The emphasis is not only on finding an algorithm to balance generalized task set but also on implementing and verifying it with simulations as well as with hardware tests.
Acknowledgment

I am grateful to my ASML supervisor Erik Niessen, for his constant help and support during my thesis work at ASML. I would also like to thank my team leads, Gertjan Schouten and Niels Sterrenburg for their suggestions, contributions and feedback during my project work. It was a pleasant experience to work within the computer systems group at ASML, Veldhoven. This work has enriched me technically as well as personally.

I also want to thank my university supervisor Dr.ir. Reinder J. Beil for his encouraging support and useful discussions regarding all aspects of this thesis. I am very much grateful for the time he spent reviewing my documents and helping me to write them appropriately.

My special thanks go to the Henk-Bodt scholarship program for supporting my study in The Netherlands and giving me the opportunity to work at ASML. I would like to thank Irene Kroon and all internship coordinators for making work at ASML more enjoyable.

I also would like to thank all my colleagues at ASML for their technical support and creating a great working atmosphere.

Finally I would like to thank my family and friends for their blessings and good wishes during my entire study period.
## Contents

1 Introduction 9  
1.1 TWINSCAN NXT:1950i 9  
1.2 Goal 10  
1.3 Problem statement 10  
1.3.1 Original problem statement 11  
1.3.2 Revised problem statement 12  
1.4 Approach 14  
1.5 Overview 14  

2 Domain Analysis 15  

3 Platform Model 17  
3.1 Hardware 17  
3.2 Software 19  
3.3 Platform characteristics 19  

4 Application Model 21  
4.1 Directed Acyclic Graph 21  
4.2 The main host tasks 22  
4.3 Analysis of the main host task graphs 25  

5 General task sets 27  
5.1 Task characteristics of general task sets 27  
5.2 Task relations 27  
5.3 Generalized model 29  

6 POOSL Model 30  
6.1 Operating System 31  
6.2 Processor 31  
6.3 Combining with TPTview 32  
6.4 Generating POOSL Model 33  

7 Task mapping algorithm 34  
7.1 Algorithm for fixed communication delay (core mapping algorithm) 35  
7.1.1 Earliest time first with capacity 35  
7.1.2 Modified critical path with capacity 36  
7.1.3 Modified Modified Critical Path (MMCP) 37  
7.2 Algorithm for data dependency 38
8 Results and Validations

8.1 Fixed communication delay .............................................. 40
  8.1.1 Taskset-I ............................................................. 41
  8.1.2 Comparison between Linux and Solaris Scheduler ............ 43
  8.1.3 Problems with preempt_rt Scheduler .............................. 44
  8.1.4 Critical path at higher priority .................................. 45
  8.1.5 Limitation of static mapping ..................................... 46
  8.1.6 Taskset-II ............................................................ 47

8.2 Data Dependency ........................................................... 49

9 Conclusion

9.1 Results achieved .......................................................... 54
  9.1.1 Original problem statement ....................................... 54
  9.1.2 Revised problem statement ...................................... 54

9.2 Future work ................................................................. 55
List of Figures

2.1 System overview ............................................. 15
2.2 System Working ............................................. 16

3.1 Resource model ............................................. 17
3.2 Host Model ................................................. 18
3.3 Intel Xeon E5540 Cache architecture ......................... 18

4.1 Measurement Phase ........................................... 23
4.2 Exposure phase until processing of First wafer .............. 24
4.3 Exposure phase per die ....................................... 25

5.1 Task relation example ......................................... 28

6.1 POOSL Model Overview ....................................... 30
6.2 Transmitting signal on Throughput port ...................... 33

7.1 Algorithm Concept ............................................ 35
7.2 ETF algorithm mapping ....................................... 36
7.3 Modified Modified Critical Path (MMCP) ...................... 37
7.4 Algorithm concept ............................................. 38

8.1 task set I .................................................... 41
8.2 Simulation of POOSL Model of task set I .................... 42
8.3 Part of static mapping schedule for task set I ............... 42
8.4 Part of mapping generated by the Linux Scheduler for task set I 42
8.5 Solaris Scheduler ............................................. 43
8.6 Wind-river Linux Scheduler .................................... 43
8.7 Excessive context switching .................................... 44
8.8 Performance deterioration when application is run with higher priority ........................................ 44
8.9 schedule for mapping generated by algorithm ................ 45
8.10 schedule for the Linux scheduler when critical path is given Higher priority ...................................... 45
8.11 Schedule with static mapping with fewer cores than required on Linux ............................................. 46
8.12 Schedule of the Linux Scheduler with fewer cores than required ........................................... 47
8.13 task set II ................................................... 47
8.14 Simulation of POOSL Model of task set II ............... 48
8.15 Part of static mapping schedule for task set II ............ 48
8.16 Part of mapping generated by the Linux Scheduler for task set II
8.17 Taskset-III
8.18 Simulation of the POOSL Model of task set III
8.19 Mapping on Host-1
8.20 Mapping on Host-2
1 TPTview
2 Screen shot of input csv file
3 Taskset IV
4 Simulation with MCP with capacity algorithm
5 Simulation with MMCP Algorithm
6 Limitation of host mapping
7 Taskset V
8 Simulation of Taskset V
9 Taskset VI
10 Simulation of Taskset VI
11 Taskset-VII
12 Mapping generated by original MCP for task set VII
13 Mapping generated by MMCP for task set VII
14 SHEsim Application Model
15 SHEsim model of single Task
16 Algorithm to create POOSL model
17 sched_switch output
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Characteristics of the TWINSCAN NXT:1950i</td>
<td>10</td>
</tr>
<tr>
<td>1.2</td>
<td>Core numbering</td>
<td>13</td>
</tr>
<tr>
<td>5.1</td>
<td>Characteristics of execution of the task set in Figure 5.1</td>
<td>29</td>
</tr>
<tr>
<td>8.1</td>
<td>Mapping of task set I</td>
<td>41</td>
</tr>
<tr>
<td>8.2</td>
<td>Mapping of task set III over a two host system</td>
<td>50</td>
</tr>
</tbody>
</table>
## List of abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALAP</td>
<td>As Late As Possible</td>
</tr>
<tr>
<td>ASML</td>
<td>Advanced Semiconductor Materials Lithography</td>
</tr>
<tr>
<td>ATCA</td>
<td>Advanced Telecommunications Computing Architecture</td>
</tr>
<tr>
<td>BNP</td>
<td>Bounded Number of Processors</td>
</tr>
<tr>
<td>CT</td>
<td>Critical path Time</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>ETF</td>
<td>Earliest Time First</td>
</tr>
<tr>
<td>HT</td>
<td>Hyper Threading</td>
</tr>
<tr>
<td>MCP</td>
<td>Modified Critical Path</td>
</tr>
<tr>
<td>MMCP</td>
<td>Modified Modified Critical Path</td>
</tr>
<tr>
<td>POOSL</td>
<td>Parallel Object-Oriented Specification Language</td>
</tr>
<tr>
<td>wph</td>
<td>Wafers Per Hour</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

ASML is one of the world's leading providers of lithography systems for the semiconductor industry, manufacturing complex machines that are critical to the production of integrated circuits or microchips. In these machines, patterns are optically imaged onto a silicon wafer that is covered with a film of light-sensitive material (photo resist). Lithography is a process, used in these machines to selectively remove parts of the film. This procedure can be repeated several times on a single wafer. The photo resist is then further processed to create the actual electronic circuits on the silicon. The optical imaging, that ASML's machines deal with, is used in the fabrication of nearly all integrated circuits [19]. Out of different kinds of machines the Twinscan lithography platform caused a revolution, when it was first launched by ASML in the year 2000 because of its higher throughput. These machines are still in a phase of constant upgrading to achieve higher throughput and new improved versions of these machines are getting launched. Also, accuracy of these machines is increasing as they work with smaller and smaller critical dimensions and overlay. ASML also manufactures extreme ultraviolet lithography machines that use 13.5 nm wavelength light. In April 2009, the Belgian research center Imec presented the world's first functional 22 nm CMOS SRAM memory cells made using ASML's prototype EUV lithography machine. This thesis will not focus on these machines as they are still in a development phase. It focuses on the Twinscan machines only.

1.1 TWINSCAN NXT:1950i

The Twinscan machine processes two wafers simultaneously. These wafers go through two main phases of processing in the machine viz. measurement and exposure. The measurement phase, as the name suggests, deals with measuring the wafer. The exposure phase does some measurement corrections and exposes the required pattern onto the wafer. The latest Twinscan machine launched in December 2008, is the TWINSCAN NXT:1950i. This machine consists of various subsystems which are running on different platforms. There are some specific subsystems that perform tasks related to the functionality while there is a dedicated subsystem performing the controlling jobs. Characteristics of the TWINSCAN NXT:1950i are given in Table 1.1.
1.2 Goal

The Twinscan NXT lithography machine built by ASML consists of two main parts, viz. embedded part and main host part. The embedded part is responsible for functionality and interaction with sensors and actuators while the main host part performs controlling tasks. This thesis focuses on the main host part. The main host has a single multicore host running Solaris. Until recently, ASML used to switch to the cores having higher frequency in order to increase the speed of the Twinscan machines. But now, focus is shifting to adding multiple, multicore hosts having a slower core frequency to reduce the power consumption. So the main host is also getting migrated to multiple hosts having multiple cores. The operating system is also getting switched from Solaris to Wind River Linux. The goal of the thesis is to find a feasible mapping of the tasks running on the main host such that the throughput of the system is improved if possible and can be made predictable. Currently, the mapping of a task on a core of the host is decided by the Solaris scheduler. The scheduler performs the global scheduling which is dynamic and can potentially introduce unnecessary resource sharing between the tasks. Though the scheduler performs the load balancing, migrations due to dynamic schedule, may invalidate the core cache, increasing the task execution time [1]. This will affect the throughput of the system adversely. Also the operating system scheduler cannot take into account the processing power available on another host to create a schedule. The same problems are observed for the Linux scheduler. Hence the goal of the thesis is to provide a mechanism which will eliminate the unrequired resource sharing and make use of the available distributed processing power thereby increasing the throughput of the system under consideration.

1.3 Problem statement

As stated, the aim of the assignment is to map the main host tasks on available cores such that the throughput of the machine is improved if possible. Currently the problem is the main host application does not use the available cores effectively which will be explained in the original problem statement. After analyzing the main host application, it is seen that it is mostly sequential and very little improvement in the throughput can be obtained by mapping such an application. But in the future, ASML have to introduce as much parallelism in it as possible in order to obtain the further improvements in the throughput. After making the application parallel, mapping it on available cores is necessary to obtain the potential improvement in the throughput. Hence in the revised problem statement, generalized task sets having parallelism are considered.

<table>
<thead>
<tr>
<th>Lens</th>
<th>Overlay</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numerical Aperture</td>
<td>Resolution</td>
<td>16 pt alignment</td>
</tr>
<tr>
<td>0.85–1.35</td>
<td>38 nm</td>
<td>2.5 nm</td>
</tr>
</tbody>
</table>

Table 1.1: Characteristics of the TWINSCAN NXT:1950i.


1.3.1 Original problem statement

There is more than a linear growth in computational load of the main host software. This has two main causes. Firstly, the embedded part is continuously improved to achieve a higher throughput, implying less time to perform the control on the main host. Secondly, the accuracy of the embedded part is increasing, implying an increase in the amount of time needed for and complexity of the main host software. Some part of the main host software is real-time, i.e. certain tasks need to be completed within a specified time frame. Although a delay in completion of these critical real-time tasks will not result in erroneous behavior, it may have a negative effect on the throughput.

In a Twinscan NXT machine, there are hundreds of tasks running on the main host subsystem. Out of these tasks there are some real-time tasks which are on the critical path when it concerns throughput. The main host part and the embedded part run in parallel and they take comparable execution time. Hence, the throughput of the machine is bounded by both the embedded part and the main host part.

The main host software is currently running on a single multi-core processor and mapping of the tasks onto the cores is handled by the operating system’s scheduler. Currently, multiple tasks may be scheduled on the same core, along with other non-critical tasks. This may cause a delay in completion of one or more, time critical tasks which in turn affects the throughput of the system adversely. An additional challenge is that the host platform will be changed to a multiple multi-core main hosts connected via a TCP/IP network to support the growing computational needs. The operating system’s scheduler running on a host can not consider the cores present on another hosts for generating the mapping. The new platform will also have hyper threading which allows one physical processor core to present two logical cores to the operating system, which allows it to support two threads at the same time[11]. But hyper threading is assumed to be disabled for this thesis.

So the actual problem is that the operating system’s scheduler is not using the available main host cores effectively. Also, when a system has tasks which are competing for the processor core at the same time, the addition of a new processor core should result in fewer collisions between the time critical tasks. Currently, there is no guarantee that this will be achieved as the mapping is handled by the scheduler. Hence, currently the main host software is not scalable with the hardware.

This thesis is concerned with creating a static mapping of the real time part of the main host software on a multiple multi-core main hosts platforms. A new mapping of the tasks is created every time the machine restarts when there are some changes in either the main host task characteristics or the main host platform characteristics or both. The assumptions made about the system are as follows.

Assumptions:

1. The main host platforms are homogeneous.

2. The main host platforms are multi-core running Wind River Linux kernel 2.6.34 with a preemptive real-time scheduler.
3. Hyper threading is considered to be disabled.

4. Time of communication between any two main host platforms is the same.

5. The main host platforms are connected via Ethernet communicating via TCP/IP protocol.

6. The task set can be represented using DAG with vertices representing the tasks and edges representing either precedence relation or data dependency.

7. The dependencies amongst the tasks and execution times of the tasks are already known.

**Deliverables:**

1. Algorithm to create static mappings of real-time tasks to cores of the multihost main host system.

2. Proof of concept of the algorithm on an available multicore multihost, main host system at ASML.

3. Document with overview of considered scheduling approaches.

**1.3.2 Revised problem statement**

During the project it was found that the main host task graph has a sequential behaviour. The majority of the tasks execute sequentially. The measurement tasks running on the main host are completely sequential while in case of exposure tasks running on the main host, only 1.6% of the total time can be saved if available parallelism is exploited. Hence the static and dynamic mapping will give very similar results in terms of the throughput. So ASML is considering making the host code multithreaded, allowing them to run in parallel. But as of now most of the main host code is not yet multithreaded. Also, as the embedded part and the main host part interact with each other, making only the main host part faster will not improve the overall throughput as the embedded part is also throughput limiting.

For this reason, a generalized application model will be created preserving the characteristics (e.g. dependency types, DAG representation) of the original task set but has parallelism to exploit the processing power of the available hosts. The longest path in a task set (considering execution time and communication time together) is the 'critical path' and the time at which the last task of the critical path should ideally be finished is the 'critical path time'. All timing measurements are relative w.r.t the release time of the first task in the task set. A mapping should be generated such that the execution of the complete task set should finish within the critical path time. It is assumed that the tasks on the critical path are always mapped on core-1 of host-1. This core is numbered as core number 1. The nomenclature used to number the cores for example, for a system having 2 hosts and 2 cores per host is shown in table 1.2.
So the problem at hand becomes a job scheduling problem which can be formulated mathematically as follows.

Given the number of hosts and cores per host find a mapping if available such that equations (1.1) and (1.2) are satisfied.

\[
CT = \sum_{i \in T_1, k \in T_1, i \neq k} (e_i^1 + c_{ik}^{11}) \text{ (if } T \text{ has more than one task)}
\]

\[
\text{or}
\]

\[
(e_i^1) \text{ (if } T \text{ has only one task)}
\]

(1.1)

and

\[
\forall j = 2 \sum_{i \in T_j} \sum_{k \in T, i \neq k} (e_i^j + c_{ik}^{jm}) \leq CT
\]

(1.2)

where,

- \( n \) is the total number of cores (number of hosts * number of cores per host),
- \( T_j \) represents the set of tasks mapped on core \( j \),
- \( T \) represents the total task set,
- \( e_i^j \) is the execution time of the task \( i \) on processor \( j \),
- \( c_{ik}^{jm} \) is the communication delay between task \( i \) and \( k \) mapped on core \( j \) and core \( m \) respectively (\( j \) and \( m \) can be the same). If the tasks are not directly connected, communication cost between such tasks is assumed to be zero,
- \( CT \) is the critical path time.

It is assumed that the load of the background processes is already known and these processes run on dedicated processor cores which are not considered part of the ‘\( n \)’ cores mentioned above or in the gaps generated by the static mapping. This thesis does not deal with allocating the background tasks in such gaps.

The revised problem is basically a multihost scheduling problem and as explained in Section 1.3.1, it cannot be solved by the operating system’s scheduler as the latter will only consider the tasks mapped on a single host for scheduling. To reduce complexity, this problem statement is decomposed into two problems.

1. At first, the communication between the tasks is assumed to be fixed irrespective of the core on which the tasks are mapped.
2. The next phase considers the Ethernet communication between the tasks having data dependency and mapped on the different hosts.

The assumptions about the system are identical to those in Section 1.3.1

Deliverables:

1. Algorithm to generate mapping for generalized task set represented using a DAG.

2. Proof of concept of the algorithm on an available main host system at ASML.

3. Document with overview of considered scheduling approaches.

1.4 Approach

The approach considered for solving the original problem statement is creating a static deployment/mapping of the tasks. The DAGs representing the tasks dependencies are created as a first step. The vertices of DAGs have execution times of the tasks while communication cost (time of data dependency) is represented along the edge. The longest path in a DAG (considering execution time and communication time together) is a critical path and the time at which the last task of the critical path should ideally be finished is the critical path time.

It was seen from the constructed DAG, that the tasks are sequential so there is hardly any opportunity to improve the throughput by adding more cores. Given the fact that ASML aims at refining their software to improve parallelism, generalized task sets were considered in the revised problem statement.

An algorithm is developed to generate this mapping for a generalized task set such that the complete task set is executed within the critical path time. The migration of a task from one core to another is avoided by setting the core affinity for the task. So unnecessary migrations can be avoided if the proper static mapping of the tasks is created. The generated mapping is tested using a POOSL model simulation. The mapping algorithm has also been verified by creating the similar set of tasks as the considered task set and running that set on real hardware with assigned core affinity as generated by the algorithm.

1.5 Overview

The remainder of the report is organized as follows. Chapter 2 gives an overview of the domain analysis and system working. Chapter 3 explains the platform model in detail. Chapter 4 describes the DAG constructed after analyzing the current tasks on the main host while Chapter 5 explains the task characteristics, task relations and generalized application model. The characteristics of the application and platform explained in these chapters are then used to create the POOSL model. Chapter 6 describes the POOSL model used for simulating the mapping and how the simulated results are combined with TPTView for analysis. Chapter 7 explains the algorithm which generates the mapping for a generalized task set. The results and validation of the generated mapping is provided in Chapter 8. At the end, conclusion and future work is given in Chapter 9.
Chapter 2

Domain Analysis

A simplified overview of the available resources in a Twinscan machine is shown in Figure 2.1. As seen, the machine is divided into the main host part and the embedded part. These subsystems communicate with each other to perform the lithography operation. The main host part currently consists of Solaris operating system running on a single multicore SPARC processor while the embedded part contains VxWorks systems, Linux systems running on ATCA (Advanced Telecommunications Computing Architecture) boards, actuators and sensors. As seen in Figure 2.1, VxWorks systems and ATCA systems get data from sensors and interact with actuators to perform functionality-related tasks.
But as this thesis is concerned with the tasks performed on the main host system, the embedded part is not discussed in detail. The main host part is responsible for controlling, computation and metrology related tasks. It communicates with the embedded part. As shown by the dotted line, the main host part is being migrated from Solaris to Wind River Linux kernel 2.6.34.

![Diagram](image)

Figure 2.2: System Working

Figure 2.2, explains the working of the system and different phases along with their timing requirements. There are two wafers present in the system at a time as shown in Figure 2.2. Processing of dies on a wafer is done sequentially. The wafer holder in the machine is called a chuck. The processing of a single wafer goes through the following sequence. The cycle starts with load/unload of the wafer. The duration of the tasks related to load/unload are bounded by mechanical activities. Then the wafer moves on to measurement-1 phase. Then the chucks are swapped and the wafer is passed for exposure after performing some measurement corrections in the Measurement-2 phase. Then the wafer is exposed to radiation in the exposure phase.
Chapter 3

Platform Model

3.1 Hardware

Chapter 2 gives a system overview of an Twinscan machines. This section will zoom in to the structure of the hardware of the main host part. The main host part running Solaris consists of just one multicore SPARC host. As explained in Chapter 2, the main host operating system will be shifted to the Wind River Linux. The platform is also shifted from SPARC to Intel Xeon E5540. Hence, in the later sections the main host is considered to be running the Wind River Linux 2.6.34 kernel on the Intel Xeon E5540 platform.

There will be more than one host running together to constitute the main host, connected via Ethernet capable of transmitting up to approximately 1000Mb/s (125 MB/s). This is shown in Figure 3.1. As of now, there will be only two hosts present in the system. But in future depending on the load, more hosts can be added. The algorithm creating the static mapping should have provisions to take addition of hardware into account.
Figure 3.2: Host Model

Now let's focus on the architecture of a single Intel Xeon E5540 processor. Figure 3.2 shows a schematic of the processing units available per host. It is a four core processor. Each core has two levels of dedicated cache (L1 and L2) and one shared cache (L3) as shown in Figure 3.3. The motherboard used in the system is an Intel Server Board S2600C0 family. It supports 8 GB of main memory with a maximum transfer rate of approximately 12800 MB/s [20].

Figure 3.3: Intel Xeon E5540 Cache architecture

The Intel Xeon processors have a special feature called Hyper-Threading (HT). Intel HT Technology allows one physical processor core to present two logical cores to the operating system, which allows it to support two threads at the same time. The key hardware mechanism underlying this capability is an extra architectural state supported by the hardware. The execution pipeline of processors based on Intel® Core™ microarchitecture is four instructions wide, meaning that it can execute up to four micro-operations per clock cycle. Intel HT Technology improves performance through increased instruction level parallelism by having two threads with independent instruction streams, eliminating data dependencies between threads and increasing utilization of the available execution units. [11]. This is shown in Figure 3.4.
While Intel HT Technology improves thread-level parallelism, the two logical processors in each physical processor core share most execution resources. Therefore, to take advantage of Intel HT Technology, an application must be able to launch additional threads in order to generate additional parallelism[11]. For simplicity, the applications considered does not take threads into account, so for this thesis Hyper threading is considered to be disabled.

3.2 Software

The operating system used by the main host part is the Wind River Linux. The following points describe the scheduling rules of wind-river Linux [21]

1. Scheduler is preemptive.

2. Priorities of the tasks are “static”. Real-time tasks have priorities ranging from 1(lowest) to 99 (highest) [8, 9].

3. The scheduling policy that is used for real-time task having same priority is “Round Robin”. For non-real-time tasks, it is “sched_other”. “sched_other” is like round robin but the time slice given to the task depends on the nice value of the non real-time task. Scheduler used by real time tasks is Linux “RT_scheduler” and by non real-time tasks is “Completely Fair Scheduler (CFS)”[8, 9].

3.3 Platform characteristics

The platform can be characterized as follows based on the assumptions explained in Section 1.3.1 and previous explanation in this chapter

1. The hosts are homogeneous (hardware and software wise).

2. A cost of the communication is considered to be the same between any 2 Hosts.
3. Hyper Threading is considered to be disabled.

4. The scheduler used is “preemptive.rt” with round robin scheduling.

5. The hosts are connected via high speed Ethernet having maximum bandwidth of 125 MB/s.

6. The main memory speed is 12800 MB/s.
Chapter 4

Application Model

The application model will represent the characteristics of a task set. As the goal of the assignment is to improve the throughput of the machine, it is important to understand the behavior of the tasks on an actual system and their characteristics. This chapter will describe how the tasks are analyzed, how the application model is created and generalized.

A task represents a single process running on a core. The tasks are analyzed using a tool developed by ASML called Throughput viewer along with ASML documents and the task code. The throughput viewer is explained in Appendix I. Based on the information gathered, directed graphs are created which represent the dependencies between the tasks. The task running on main host can also depend on the task running on the embedded part. These dependencies and characteristics of the tasks are explained in following subsections. As actual application turns out to be sequential, generalized task sets having similar characteristics as actual application model are created.

4.1 Directed Acyclic Graph

A directed acyclic graph (DAG) is a graph with no directed cycles. It consists of nodes and directed edges, each edge connecting one node to another, such that there is no way to start at some node \( v \) and follow a sequence of edges that eventually loops back to \( v \) again [15].

An application is represented by a directed acyclic graph (DAG) \( G = (V, E) \), where \( V \) is a set of nodes and \( E \) is a set of directed edges. A node in the DAG represents a task which has to be executed on some processor core. The task can be preempted by a higher priority task on the same processor core. The weight of a node is called the execution cost (time in this case). The edges in the DAG, represent dependencies among the nodes. The weight of an edge is called the communication cost of the edge. The source node of an edge is called the predecessor node while the sink node is called the successor node. If there are more than one predecessors then a predecessor having highest communication cost on its connecting edge is called 'critical predecessor'. A node with no predecessor is called an entry node and a node with no successor is called an exit node.

There can be two types of dependencies between the tasks. The tasks can
either have precedence constraints with some fixed time between the completion of one task and the start of another or there can be some communication between the tasks. In later case, amount of data to be communicated is mentioned along the edge. There are some sequence of tasks which are performed more than once which is highlighted with a box around those tasks. The number of iterations are mentioned inside the box.

Whenever application is modeled using a DAG, the execution time and communication time between the tasks can be easily determined. A DAG based application can be mapped on the multihost, multicore platform by

(i) assigning tasks to Hosts.

(ii) determining how to map the tasks in each host on multiple cores [23].

While addressing (i), overheads due to data communications among connected tasks must be considered since the communication across the hosts is slower and should be minimized as far as possible. While addressing (ii), available processing power on a host should be utilized as much as possible. Hence the execution times of the tasks should be considered.

Hence, it is possible to separate the multihost mapping and multicore mapping using a DAG based application model.

4.2 The main host tasks

After analysis of the tasks running on the main hosts and discussing with the execution architects at ASML, a subset of the tasks running on the main host is created such that the subset will contain the tasks which may have some parallelism or which are already multithreaded. The graphs showing the execution sequence of various tasks in this subset are created. They are shown in Figure 4.1 through 4.3.
Figure 4.1: Measurement Phase
Figure 4.2: Exposure phase until processing of First wafer

Nodes of the tasks running at highest priority are indicated in **bold**
In the above figures, each node represents a task. Most of the tasks run as a single process on the processing platform. Whenever a task is multithreaded, it is explicitly mentioned. Execution times of the task are mentioned inside the task node. The communication delay is mentioned along the arrow. If certain tasks are performed in a loop then it is displayed with a box around those tasks and the loop count is mentioned inside the box.

If one task is dependent on another through either a precedence relation or a data dependency (not visible in above graphs) then the arrowhead is pointed at the dependent task. The main host tasks may not be directly dependent on each other but there may be embedded task(s) between them causing these tasks to be dependent on each other indirectly. But as this thesis is concerned with the main host part, only the main host tasks analyzed. Therefore it cannot be stated which tasks are dependent indirectly via embedded task(s).

As these graphs resemble the description of a DAG as given in 4.1, it can be concluded that the tasks running on main host can be represented using a DAG. It should be noted that the graphs shown in Figures 4.1 through 4.3 represents the application model for processing a single wafer and this complete application repeats in a loop for processing the subsequent wafers.

### 4.3 Analysis of the main host task graphs

Consider the Measurement phase graph shown in Figure 4.1. It can be seen that this graph is completely sequential with no parallelism in it. The critical path time of the considered task graph is 9398 ms while time spent on actual execution is around 239 ms (only 2.5% of total critical path time).

Now consider the graphs in 4.2 and 4.3. The critical path time for the exposure phase based on this subset of tasks is around 3141 ms. The amount of actual time spent in executing the tasks on critical path is around 156 ms (around 5% of total critical path time). The exposure phase shows some parallelism. The total time saved if all the available parallelism is exploited is around 50 ms. So only around 1.6% of exposure graphs can run in parallel.

To find out the multithreaded tasks present in the system, discussions are
done with the execution architects at ASML. It was concluded that currently, most of the tasks run as a single process and ASML is still in a phase of making the tasks multithreaded.

Most of the tasks (not represented with bold nodes) are running at the same priority which is just below the system priority. The priorities are already decided by ASML and this thesis does not make any alterations in these priorities.

Also it should be noted that the main host tasks also communicate with the embedded part. Hence making the main host part faster alone, may not improve the throughput of the system as the main host tasks will still have to wait for the embedded part. The long wait time before the 'TE-4' or 'TE-10' shown in Figure 4.2, is probably due to communicating with the embedded part.

As there is very little parallelism in the task set and application is mostly communication bound, it may be possible to map these task graphs on single host having two cores with one core dedicated for the measurement and one core dedicated for the exposure phase. But the feasibility of such mapping is beyond the scope of this thesis as it requires analysis of all the tasks running on the main host and tasks running on the embedded part which communicate with the main host and also, the specifications about the embedded and the main host part communication. This thesis from this point onwards considers the generalized task sets and the following subsection will discuss the tasks characteristics extracted which are useful in creating generalized task sets.
Chapter 5

General task sets

This chapter explains the task characteristics, task relations and how generalized task sets are modeled.

5.1 Task characteristics of general task sets

As seen from the main hosts tasks, the measurement part is completely sequential while only 1.6% of exposure graphs can be executed in parallel. However, following characteristics can be extracted from the graphs shown above and assumptions mentioned in section 1.3.1.

1. Tasks can be represented by a DAG.

2. There is either a precedence relation between the tasks or a data dependency (not visible in the graphs but code analysis suggests that there are data dependencies present). Some dependencies may be indirect because of the communication with the embedded part.

3. The dependencies amongst the tasks and execution times of the tasks are already known.

4. Most of the tasks are still not multithreaded, hence multithreaded tasks are not considered in generalized task sets.

5.2 Task relations

In order to find the mapping of the tasks on cores which satisfies equations (1.1) and (1.2) it is required to define the relations between the tasks. These relations are then used in the mapping algorithm. These relations are defined using the earliest release time and earliest finish time or using the structure of the DAG. The earliest release time of a task is the earliest time a task can start its execution given that all the predecessors have finished their execution and required fixed delay time or communication has taken place. The earliest finish time of a task is equal to the addition of the earliest release time and the execution time of the task. If the earliest release time and the earliest finish time of all the tasks are known, then it is possible to determine the inevitable
gaps in the schedule due to communication. Mapping algorithm should then try to map some other tasks in these gaps if possible.

As task set is represented using a DAG and the execution times and dependencies amongst tasks are known, it is easy to find the earliest release time and earliest finish time. There can be one of three types of relations between the tasks. These relations are defined as follows.

1. **Dependent**: If there is a precedence constraint or a data dependency between two tasks then the successor tasks are said to be “dependent” on the predecessor. A task is dependent on all the tasks on which its predecessor is dependent. Hence, dependency is transitive and can be concluded from the structure of the DAG.

2. **Independent**: If tasks are not dependent on each other then they are Independent.

3. **Independent_overlap**: This is special case of independent relation. Suppose two tasks “task A and task B” are independent. The earliest release and finish time of task A is represented as “\(R_A\)” and “\(F_A\)” respectively. Similarly, the earliest release and finish time of task B is represented as “\(R_B\)” and “\(F_B\)” respectively. Then these two tasks are said have independent_overlap relation if one of the following equations is satisfied.

\[
\begin{align*}
R_B & \leq R_A \leq F_B \quad (5.1) \\
R_B & \leq F_A \leq F_B \quad (5.2) \\
R_A & \leq R_B \leq F_A \quad (5.3) \\
R_A & \leq F_B \leq F \quad (5.4)
\end{align*}
\]

The tasks under independent_overlap category can run in parallel. So the mapping algorithm should try to map them on different cores as far as possible as if mapped on same core, they will execute in round robin manner which may increase the overall finish time of the application.

These relations can be best explained with a simple task set as shown in Figure 5.1.

![Figure 5.1: Task relation example](image)
Table 5.1: Characteristics of execution of the task set in Figure 5.1

<table>
<thead>
<tr>
<th>Task</th>
<th>Earliest release Time (ms)</th>
<th>Earliest finish Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>110</td>
<td>210</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>80</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td>60</td>
</tr>
</tbody>
</table>

The earliest release and finish times of the task set are shown in Table 5.1. Based on these characteristics, three relation categories are defined between the tasks.

1. Task2, Task3, Task4 are “dependent” on Task1
2. Task3 and Task4 are “independent” with Task2
3. Task3 and Task4 are under “independent_overlap” category. This due to the fact that, earliest release time of the Task4 is greater than earliest release time of Task 3 and less than earliest finish time of Task3. Hence the mapping algorithm should try to map these tasks on different cores if possible.

5.3 Generalized model

As stated earlier, the tasks running on the main host do not show much parallelism. So most of the tasks have to be/can be performed sequentially. For this reason generalized task sets are created such that if not mapped properly on available resources, these task sets will exceed the critical path time. The characteristics mentioned in Section 5.1 are preserved while creating a generalized task set. Also as the embedded part is out of scope of this thesis, information about exact dependencies between the main host and the embedded tasks can not be obtained. Hence these dependencies are not modeled in generalized task model. But if the exact time say $t_c$, between finishing of the predecessor main host task and starting of the indirect successor main host task is known, it is possible to eliminate the embedded part dependency and model the dependency as precedence relation between the main host tasks with weight of the edge between them equal to $t_c$. 

29
Chapter 6

POOSL Model

The Parallel Object-Oriented Specification Language (POOSL) is a modeling language which is used to develop models of hardware/software system for analysis purposes[16]. The POOSL model is also very useful in analyzing timing characteristics of the software. Another nice feature of POOSL is that details of an application model execution can be logged to a file which can be opened and analyzed using Throughput viewer (TPTview). The TPTview tool is explained in appendix I. Figure 6.1, shows an overview of how POOSL models of application, platform, TptView and simulationController are arranged. As seen, models communicate with each other by sending a message over a dedicated channel. The SimulationController decides the simulation time of a model. The POOSL model of the throughput viewer will create a file which can be then read using the TPTview tool. The model shown in Figure 6.1 is constructed using a simulator tool called “SHESim”.

Figure 6.1: POOSL Model Overview
A detailed description about the POOSL models and its generation is given in Appendix IV. Following subsections will only describe the important parts of the POOSL model generated.

### 6.1 Operating System

The thesis focuses on Wind River Linux operating system which has preemptive round-robin scheduler as stated in Section 3.2. Such a scheduler will preempt the current running task if there is a higher priority task waiting for the core. If there are more than one task waiting for the core having a same priority then they will be scheduled in a round robin manner with specified time interval. The time interval is dependent on the task priority. For the modeling purpose, a constant time slice (10 ms) is considered. Algorithm 6.1, show how operating system schedules the task.

**Algorithm 6.1 operating system**

```plaintext
if NewTask then
  register_task () //add the task to run queue according to priority
  if next_task_to_schedule == NewTask then
    if processor_not_busy then
      execute NewTask
    else
      preempt CurrentTask
      execute NewTask
  else if task_execution_finished then
    schedule next task if available
  else if round_robin_interval expires then
    schedule next task if available

6.2 Processor

The processor Model will be used to simulate the execution behavior of the task. The behavior which has to be modeled includes the following points:

1. If the currently executing task is being switched, delay for context switch time. (Currently the specification are not known so this feature is added but not used)

2. If the task is getting executed for the first time then allocate the memory required by that task
3. If allocated time slice (10 ms for this thesis) is expired, signal operating system and request for next task.

4. When a certain task is being executed constantly monitor the preempt signal from the operating system.

The execution times of tasks on the processor are given to this model via a text file.

6.3 Combining with TPTview

TPTview is a throughput analysis tool developed by ASML. This tool reads the file written in a specific format and interprets it graphically. It displays how the tasks are being executed on different processor cores. The POOSL is as already explained, a modeling language. Simulator will execute the statements written in POOSL to simulate the behaviors of the system under consideration. So, for combining TPTview with POOSL model, a file in a specific format has to be written to a proper location by simulator so that it can be read by TPTview.

The format of the file is as follows:

```
"[resource],[activity],[start],[end],[context]"
```

Where,

- **resource** (String) : represents a processor core
- **activity** (String) : represents a task executing on processor core
- **start date** : time at which the task (activity) starts/resumes its execution on the processor core represented by resource. It is measured in seconds elapsed since the starting of the POOSL simulation.
- **end date** : time at which the task (activity) finishes/stops its execution on the processor core represented by resource. It is measured in seconds elapsed since the starting of POOSL simulation.
- **context** (String) : it is used to discriminated between different cores by using different colour for each task.

ASML has a specific time format which is “\textit{dd MMM yyyy HH:mm:ss SSSSSS}”. TPTview tool will internally convert the columns representing the time into the mentioned format.

For writing the required file, a port is added to the processor model. Every time the task gets preempted, time sliced or finishes its execution a signal is transmitted on this port in a required format. Figure 6.2, shows a part of the Node model, showing this behavior.
Figure 6.1 shows a process class TPTview, which is connected with the MP-SOC platform model. This process class will accept the signal on the throughput port and append it to the proper file.

6.4 Generating POOSL Model

This thesis deals with the generalized application and platform model. So the applications and the platform model can keep on changing. Simulating using the POOSL model is very helpful as it gives the timing details about how tasks will be scheduled on an actual system. So mapping can be easily verified with the POOSL model simulation and TPTview. But creating a POOSL model for changing applications and platforms can be very time consuming. Hence there is a need to generate the POOSL model automatically. An algorithm is written and implemented for this purpose. This algorithm is explained in detail in Appendix IV. But before generating the POOSL model, mapping has to be known. An algorithm used for generating the task mapping is explained in the next chapter.
Chapter 7

Task mapping algorithm

The problem statement is divided into the two problems as stated in Section 1.3.2. The first algorithm considers application models having fixed communication between the tasks (just precedence relation). The communication cost between the tasks is assumed to be fixed as mentioned in an application model, irrespective of the host it is mapped on. So actually what this algorithm generates is the core mapping by considering all the available cores are present on the same host. In the second part, tasks having data dependencies and Ethernet communication are also considered. So a second algorithm is required for creating a host mapping such that the communication between the hosts will be as small as possible and then the core mapping on each host is generated using the core mapping algorithm. The two important terms used in these algorithms are the core capacity and the host capacity.

\[
\text{core capacity} = \text{criticalPathTime} \quad (7.1)
\]

\[
\text{host capacity} = \text{criticalPathTime} \times \text{number of cores per host} \quad (7.2)
\]

criticalPathTime is the same as CT defined in equation (1.2).

Whenever a task is being mapped on a core or host, the capacity of the core or host respectively is reduced by the amount of "execution time" of the task. So a remaining host capacity and remaining core capacity can be defined as

\[
\text{remaining host capacity} = \text{host capacity} - \text{execution times of the tasks mapped on this host} \quad (7.3)
\]

and

\[
\text{remaining core capacity} = \text{core capacity} - \text{execution times of the tasks mapped on this core} \quad (7.4)
\]

The following subsections explain these two algorithms in more detail.
7.1 Algorithm for fixed communication delay (core mapping algorithm)

![Figure 7.1: Algorithm Concept](image)

The algorithm concept and main steps executed for generating a mapping in this case are shown in Figures 7.1. The input given to the algorithm is in the “csv” format. The screen shot of such a file (opened using MS Excel) is shown in Figure 2 in appendix II. Based on the input ‘csv’ file, timing calculations to find the earliest release and finish time and the critical path calculations are performed. Then the task mapping algorithm is used to generate a mapping. Following subsections describe the mapping algorithms being considered.

After a mapping is generated, a POOSL model is automatically created and it is simulated using ‘rotational’. The ‘rotational’ is a high-speed execution engine for simulating the POOSL models. How the POOSL model is created from the generated mapping is explained in appendix IV. The simulation is linked with TPTview as explained in Section 6.3. A simulation of the model is very helpful in visualizing how the task set will execute on the real hardware. Once the critical path tasks are known, different algorithms are considered for creating a mapping for the rest of the tasks.

7.1.1 Earliest time first with capacity

One of the algorithms considered is similar to “Earliest Time First (ETF)” explained in [3] but in this case mapping is generated only for non critical tasks and instead of mapping the task on the core where it can start executing as early as possible starting time, core having maximum remaining core capacity is used. This algorithm is very simple to implement. This algorithm can be described as follows.

1. Sort the tasks in ascending order of release time. Traverse the list and perform steps 2 to 4 for each task.
2. Sort the cores in descending order of remaining core capacity.
3. Find the first core where the task can be executed.
4. Map the task on that core.

![Figure 7.2: ETF algorithm mapping](image)

The problem with this algorithm is that, if tasks having large computation times are getting released later in the schedule, they might have to share the processing core with some other task prolonging their execution time beyond the critical path time. Consider the taskset-I shown in Figure 8.1. The simulation of mapping generated by the ETF algorithm is shown in Figure 7.2. As seen, Task7 is released later than Tasks 2, 4, 5 and 6. Hence it is scheduled later and have to share the processor core with other task and it therefore completes after the critical path time.

7.1.2 Modified critical path with capacity

To solve the problems with the “Earliest time first with capacity” algorithm another algorithm was studied. It maps the tasks depending on their “As Late As Possible (ALAP)” release time. The Algorithm is explained below. This algorithm is very similar to the Modified Critical Path algorithm explained in [3]. But original algorithm maps the tasks on the core where it can start executing as early as possible. But this approach does not utilize all the available processing power causing tasks to exceed critical path time. This is illustrated with an example in appendix III, task set VII. Hence, MCP is modified to use remaining core capacity instead of earliest starting time on core. A mapping is generated only for non-critical using this algorithm.

1. Sort the tasks in ascending order of ALAP release time. Traverse the list and perform steps 2 to 4 for each task.
2. Sort the cores in descending order of remaining core capacity
3. Find the first core where the task can be executed.
4. Map the task on that core.

So the tasks which has a greater execution time than the other tasks or task on which many other tasks are dependent are mapped first. But with this algorithm, there is a possibility that the tasks belonging to the “independent_overlap”
category, are mapped on same core whereas they can be executed in parallel. This is illustrated in appendix III, Taskset IV. To overcome this problem, the mapping algorithm is improved to an algorithm which is called as Modified Modified Critical Path (MMCP). This algorithm shown in Figure 7.3.

### 7.1.3 Modified Modified Critical Path (MMCP)

![Flowchart of MMCP algorithm]

In this algorithm, an additional check is added to see if it is possible to run the tasks in parallel. If it is possible, tasks are mapped on different cores otherwise mapping is generated in the same way as the Modified Critical Path with capacity. Appendix III shows an example which illustrates the benefits of additional check for independence of the tasks.

This algorithm is still not optimal as there can be some gaps remaining in the mapping generated suggesting a time is available in which tasks can be executed prior to the critical path time. But still this algorithm gives improvement over the original MCP and MCP with capacity algorithm. For the example shown in appendix III - task set VII, improvement observed over MCP in terms of finish
time of the task set is 8.5% while for the task set IV improvement observed over MCP with capacity is around 1.2%.

7.2 Algorithm for data dependency

When considering multiple hosts, deciding which tasks to run on which hosts is very important. The approach shown in Figure 7.4 is followed to create the Multihost mapping.

![Algorithm concept]

Figure 7.4: Algorithm concept

The mapping algorithm should make sure that all the tasks finish within the critical path time. Firstly, a mapping of the task set on a single host is created using the MMCP algorithm shown in Figure 7.3. For this mapping it is assumed that all the available cores are present on the same host and if the tasks are mapped on the same host then it is assumed that the tasks use the main memory for communication irrespective of core they are mapped on. In the next step, this mapping is extended to the multiple hosts. If the tasks are mapped on the different hosts then the delay corresponding to the communication between the tasks is added considering the Ethernet speed.

For example, if the system has two hosts with two cores each, then these total four cores are assumed to be present on single host at first. A mapping is generated for such a system using the MMCP and then communication costs
are modified considering the actual system (two separate hosts in this case). If this mapping still satisfies the critical task time then that is the final mapping and the algorithm stops. In case the schedule takes more time than the critical path time a new host mapping is created by following subsequent steps. This algorithm is similar to the one explained in [23] but in this thesis only the non-critical tasks are mapped using this algorithm.

1. Map the critical tasks on core-1 of the host-1.

2. Order the remaining tasks in task pairs in descending order of the communication cost. Traverse the list and perform steps 3 to 7 for each task pair if the successor is not already mapped.

3. Order the hosts in ascending order of the remaining host capacity.

4. If the predecessor in the task pair is in the critical path
   (a) Then try to map successor on host-1.
   (b) Else find the next host which has enough capacity

5. Else if the predecessor is not in the critical path and predecessor is not already mapped
   (a) Then map both the tasks on the host which has enough capacity for both the tasks.

6. Else if the predecessor is mapped try to map the successor on the predecessor’s host.

7. Else search for the host, in the order obtained in step 3, which has enough host capacity remaining to accommodate the successor.

Once all the tasks are mapped on the available hosts (using step 4, 5, 6 or 7), the core wise mapping is generated by considering the MMCP algorithm shown in Figure 7.3 for the tasks mapped on the same host.

Now check if the finish time of task set is within the critical path time. If it is not then the critical path is still not satisfied and the mapping cannot be created using this algorithm on the provided platform model such that the complete task set will finish within the critical path time. In the algorithm considered, task pairs are arranged in descending order while hosts are arranged in ascending order of the remaining capacity. Hence, this algorithm tries to map the task pairs with the highest communication cost on the same host as long as the host has capacity. But if the host does not have enough capacity to execute both tasks then such tasks might get mapped on different hosts. The algorithm tries to map the tasks with minimum communication cost across different hosts. But this communication cost can still cause the finish time of the task set to exceed the critical path time. This can occur even if the platform has enough cores to support the load but they are wrongly configured. This limitation of the host mapping algorithm is illustrated by means of an example in appendix III.
Chapter 8

Results and Validations

This chapter will present some results which will compare the performance improvements achieved due to static mapping over the mapping created by the Linux scheduler. Results are verified under two different conditions viz. fixed communication delay and data dependency. Tools used for visualizing the execution of the tasks on Wind River Linux are Ftrace, GRASP and GTKWave. A description about these tools is given in Appendix V.

8.1 Fixed communication delay

Firstly, applications which have precedence relation but do not have any data dependency are considered. The communication time between two tasks is considered to be fixed irrespective of the hosts they are mapped on. So, specific amount of delay is inserted between two tasks by allowing predecessor task to sleep before it can trigger the successor task. The type of applications considered have the following characteristics.

\[ \sum_{i \in T, k \in T, i \neq k} (e_i + c_{ik}) \leq n \times \text{criticalPathTime} \quad (8.1) \]

where,

- \( n \) is the number of cores (number of hosts * number of cores per host). \( n \) must be an integer.
- \( T \) represents task set.
- \( e_i \) is the execution time of the task \( i \).
- \( c_{ik} \) is the communication delay between task \( i \) and \( k \).
- \( \text{criticalPathTime} \) is the minimum time the task set will take to execute considering fixed communication delay. It is the same as \( CT \) defined in equation (1.2).

The static mapping will be useful only if available processing power is enough to accommodate all the tasks in the task set. Hence only the applications satisfying equation (8.1) are considered. If this limit is exceeded, then the dynamic mapping will be more efficient as it can explore the gaps present in the schedule generated.
8.1.1 Taskset-I

Consider an Application shown in Figure 8.1. The priority of all the tasks is assumed to be the same. Hence whenever more than one task gets scheduled on the same core at the same time, those task will execute in a round robin manner. It has five non critical tasks (task2, task3, task4, task5, task6) belonging to the overlap_independent category.

This is task set is provided as a input to the algorithm shown in Figure 7.3. According to equation (8.1), summation of the execution time and communication time is 2630 ms. Critical path consists of Task1, Task3, Task8, Task9 and Task10 and ideally it takes 780 ms to complete its execution. Hence the number of cores field is set to 4 as it is the smallest integer which satisfies equation (8.1).

The mapping generated by the algorithm is given in Table 8.1 and the result of the simulation of the POOSL model is shown in Figure 8.2. Task0 and Task11

---

<table>
<thead>
<tr>
<th>Task</th>
<th>Core Number</th>
<th>Earliest release time (ms)</th>
<th>Earliest Finish time (ms)</th>
<th>ALAP release time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>30</td>
<td>230</td>
<td>580</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>20</td>
<td>70</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>40</td>
<td>140</td>
<td>680</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>50</td>
<td>200</td>
<td>630</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>60</td>
<td>590</td>
<td>280</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>70</td>
<td>770</td>
<td>80</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>170</td>
<td>370</td>
<td>170</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>470</td>
<td>670</td>
<td>470</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>720</td>
<td>780</td>
<td>720</td>
</tr>
</tbody>
</table>

Table 8.1: Mapping of task set I
are additional tasks generated for POOSL model and they have negligible execution time.

Figure 8.2: Simulation of POOSL Model of task set I

Figure 8.3: Part of static mapping schedule for task set I

Figure 8.4: Part of mapping generated by the Linux Scheduler for task set I

This task set was run on hardware to compare the simulation results but as available hardware has only two cores, task6 and task7 are eliminated to
satisfy equation (8.1) On the hardware, fixed communication delay is mimicked by a sleep statement while the execution times are generated by doing some random calculations in a loop. It was seen that there are some system processes which interfere with the tasks. So the amount of time required to execute the application is little more than the expected value. But these interferences are unavoidable as they are required for proper execution of the operating system. The core affinity was set using the `taskset -c` command. After the core affinity is set, the task is allowed to run only on that core.

As seen from Figure 8.4, the Linux scheduler schedules task8, task4 and task5 on the same core. Due to this, these tasks are executed in a round-robin manner causing delay in finishing of task8. Task8 is in the critical path for this task set. Any delay in the finish time of task8 will cause a delay in the finish time of the task set reducing the throughput. This is avoided in the static mapping as shown in Figure 8.3.

### 8.1.2 Comparison between Linux and Solaris Scheduler

Consider the same application shown in Figure 8.1, but again without task 6 and task7. Also the number of cores are reduced from four to two. The inequality in equation (8.1), is still satisfied. The mapping generated by the Solaris scheduler is shown in Figure 8.5.
Non critical tasks interfere with the critical tasks on Solaris as well but still the Solaris scheduler performs better than the Linux Scheduler. The reasons for this are given in Section 8.1.3.

8.1.3 Problems with preempt rt Scheduler

Figure 8.7, shows the part of the schedule generated by the Linux scheduler. The tool used for generating this waveform is "gwkwave".

A major problem with the Linux is unnecessary context switching when the idle cores are present. As seen in Figure 8.7, task 9 is switching between the "core 0" and "core 2" every time a sirq-timer on that core is executed. This excessive context switching prolongs the finish time of the task reducing the throughput. This behavior is only observed when the scheduler selected is "preempt rt". This is due the fact that sirq-timers are automatically enabled when the scheduler selected is "preempt rt". As ASML will be using the "preempt rt" scheduler only this case is considered.

One of the possible solutions is to run the tasks at a higher priority than sirq-timer. The real-time priority of sirq-timer is 49. So the same application is run with priority 50 so that tasks won't be interrupted by the sirq-timers. But then performance is further deteriorated. The reason for this deterioration can be best explained with Figure 8.8.

Figure 8.8 shows the part of the execution of taskset-I. Here, task3 and task2 are executed on "core2" in round robin manner. At the end of task3's
execution it tries to make task8 ready for execution. But, the sirq-timer of core2 is interrupted by task2. Hence it's not possible to change the state of task8 to runnable. Hence increasing the priority of the tasks above sirq-timers is not an option as it interferes with the proper system working.

So one feasible solution of this problem is setting a core affinity as generated by the task mapping algorithm so that the excessive context switching would not take place.

### 8.1.4 Critical path at higher priority

Another solution for limiting the overall execution time to critical path time is giving maximum possible (just below system tasks) priority to critical path tasks. With this approach, it is made sure that, whenever critical path task is released it will preempt any other user task and will be completed without any interruption by other user tasks. But with this approach the problem of excessive context switching is not solved. To test this approach, two tasks are allowed to run simultaneously on two core host. Task having higher execution time has given higher priority. The unnecessary context switching is observed in this case as well when the applications is scheduled using the Linux scheduler.

![Figure 8.9: schedule for mapping generated by algorithm](image)

![Figure 8.10: schedule for the Linux scheduler when critical path is given Higher priority](image)

As seen from the “marker” field in Figures 8.11 and 8.10, the difference between execution times is very small (order of 20 micro-seconds). Such a small difference can be due to interruption due to system tasks, cache behavior etc. So just a single execution of task is not enough to prove that, setting the core affinity is better than giving higher priority to critical path tasks. Hence the same task is called sequentially and execution times were compared. When total execution
time of such a task set is around 80 seconds then with mapping around 100 milli-
seconds are saved and for 800 seconds, mapped task set performs around 900
milli-seconds faster than the schedule generated by the Linux scheduler having
higher priority to critical tasks.

8.1.5 Limitation of static mapping

The condition stated in equation (8.1), should be satisfied in order to make
static mapping work at least as good as mapping created by scheduler. If this
condition is not satisfied, scheduler mapping will outperform the static mapping.
This is because scheduler can migrate the tasks to different cores and exploit the
gaps due to communication/precedence relation present in the schedule. This
can be inferred from Figures 8.11 and 8.12. These figures show the execution
of Taskset-I on just two core instead of four. Critical path length is 780 ms and
total time of taskset-I is 2630. So inequality in equation (8.1), is not satisfied.

\[ \frac{2630}{2} \geq 780 \]

Figure 8.11: Schedule with static mapping with fewer cores than required on
Linux
As seen after comparing Figures 8.11 and 8.12, the Linux scheduler finishes execution of task set in 1350 ms and the static mapping takes around 1680 ms. Hence the Linux scheduler performs much better than the static mapping generated by the mapping algorithm.

8.1.6 Taskset-II

Figure 8.13: task set II
The algorithm is tested with another task set and similar results are obtained. Taskset-II is shown in Figure 8.13. This task set was more helpful to analyze the effectiveness of the mapping algorithm as there are many overlapping tasks.
at different instances along the critical path. Critical path consists of task1, task2, task4, task6, task10. Number of cores are set to 2. Simulation results of POOSL model are shown in Figure 8.14. Static mapping results and the Linux scheduler results are shown in Figure 8.15 and Figure 8.16 respectively.

As seen in Figure 8.16, the Linux scheduler schedules task 3 and task 9 on two separate cores and there no core available to schedule task 2 causing delay in starting of task 2. As task 2 is in the critical path, this delay causes an increase in the overall execution time of the application.

8.2 Data Dependency

This section will explain the results obtained when task sets having data dependencies is considered. In case of a multihost system, whenever there is data dependency present between the two tasks, the communication delay between those two tasks is dependent on mapping of the tasks. If the tasks are mapped on different hosts, then they have to communicate using the Ethernet connection. If these tasks are mapped on the same host, the communication will be much faster as data is present in the memory. It is assumed that the memory is sufficient to store the produced data and also collision between data transfers across Ethernet are not considered for simulation purpose. The critical path in this case is found out by assuming the tasks communicate using the main memory.

The Ethernet speed of the connection is considered to be 1000Mb/s or 125 MB/s. When tasks are mapped on the same host simulating the data dependency is difficult. The generated data can be present in cache making communication considerably fast or else it might have to access data from the main memory. Cache behavior is highly dependent on the programming style and may vary from application to application. As this thesis considers generalized applications, the data is assumed to be fetched from the main memory. The speed of the main memory on the considered hardware is 12800 MB/s [20]. Also it is assumed that the data is written while execution of the successor task and the predecessor tasks reads this data before starting its execution. Hence a delay corresponding to the reading of the data from the main memory is considered while simulating the model.

Consider a task set shown in Figure 8.17. This task set is very similar to Taskset-I shown in Figure 8.1. The main difference is now communication delay is mentioned in terms of data dependency.
The mapping generated by the mapping algorithm for this task set is shown in Table 8.2. The task set III is mapped on two hosts each having two core per hosts. The cores on host-1 are numbered as node-1 and node-2 while cores on host-2 are numbered as node-3 and node-4.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Overall Core Number</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 8.2: Mapping of task set III over a two host system

The simulation of the POOSL model generated for this task set is shown in Figure 8.18. The network speed and memory access speed assumed to be constant.
To test if the simulated results can be regenerated on Hardware, the tasks are mapped on available hardware. To generate the data dependency across the two hosts, a file of size of given data dependency is transferred over the Ethernet. To simulate the memory access, the given amount of memory is allocated into main memory and some data is copied in that memory. The available hardware is slower than the actual hardware. The memory speed obtained on available hardware is around 6500 MB/s. Also, data is copied across Ethernet using \texttt{scp(secure copy)}”. It uses encryption and hence the Ethernet speed obtained is around 60 MB/s. The results of mapping are shown in Figure 8.19 and Figure 8.20.
Figure 8.19: Mapping on Host-1

Figure 8.20: Mapping on Host-2
As seen in Figure 8.20, tasks 2, 4, 5 and 7 are monitoring if task 1 is completed. When task-1 is completed, these tasks start copying the files across the Ethernet. The time from this instant is considered to compare the time required to complete the tasks. Because of Ethernet collisions, it takes more time to copy the data, but still, tasks on host 2 completes before the tasks on host 1. This is evident from the marker field highlighted. Host-1 takes 1152 ms to complete the assigned tasks and Host-2 takes 971 ms to complete the tasks mapped on this host.
Chapter 9

Conclusion

This chapter explains the main achievements of the thesis and the future work.

9.1 Results achieved

This section is divided into two subsections. At first, the results obtained after investigating the original problem statement are mentioned. In the second subsection, results related to the revised problem statement are stated.

9.1.1 Original problem statement

It is seen from the generated task graphs that the original tasks running on the main host are mostly sequential. Only about 1.6% of the total exposure time on the main host can be saved if the available parallelism is exploited while the measurement phase is completely sequential. Also, there is communication between the main host part and the embedded part. Hence, multithreading the main host tasks or making them parallel will only make the main host task set faster but it will not affect the overall throughput as the main host part will just have to wait even longer for the embedded part to respond. Hence the main host part, the embedded part and the communication between them should be improved together to improve the throughput.

As the main host application is sequential, generalized application models having parallelism are considered in the revised problem statement.

9.1.2 Revised problem statement

The problem with the generated schedule by the operating system’s scheduler is that the finish time of the task set can not be determined. Hence an algorithm is created for generating the static mapping over multi-host multicore systems. MCP (Modified critical path) is one of the best performing mapping algorithms in a BNP (Bounded Number of processors) category for mapping DAG based application models [22]. This algorithm is improved further to reduce the overall finish time of an application model. With the generalized application model it was seen that given the number of hosts and cores per host, the algorithm tries to find a mapping if available such that the complete task set finishes within the critical path time. A problem with the operating systems scheduler is that
it cannot handle the mapping across multiple host. A static mapping generated by the given algorithm across multiple hosts, solves this problem.

Also, a problem observed with the Linux scheduler is, if there are idle cores, then a running task switches between these cores, thereby invalidating the core's cache and increasing the overall execution time of the task. This can be avoided by setting the core affinity to a task, allowing the task to run only on a single core.

If the tasks are assigned to the cores according to the mapping generated by the algorithm explained in this thesis, task sets can be guaranteed to finish within the critical path time. The gain in terms of execution time obtained due to static deployment over the operating system's scheduler is vastly dependent on the task set. If the task set is just a single sequential path then the gain will be 0%. Out of the tested generalized application models, maximum saving seen in terms of overall finish time of the application, is about 13%.

If tasks are assigned to a host beyond host capacity then the schedule generated by the operating system's scheduler takes less time to execute than the schedule generated when tasks are mapped according to the mapping algorithm.

Hardware test results obtained are very similar to the POOSL model simulations for tested generalized applications. It was seen that execution on hardware takes more time than simulation. The difference in finish time observed is around 2% to 12%. This change is observed as simulation does not consider the background processes on real hardware, context switches due to system processes, deviations in the memory and the Ethernet speed etc.

9.2 Future work

In this thesis, some assumptions are made to simplify the modeling, simulating and mapping generation. To make the simulations and mapping more realistic in the future, models can be built with weaker or different assumptions.

Hyper-threading is considered to be disabled in this thesis. Almost all Intel processors come with hyper-threading nowadays. Considering hyper-threading can increase the number of schedulable task sets. Also all the hosts are considered to be homogeneous. Considering heterogeneous hosts can be a next step.

When mapping on the same host is considered, modeling the cache behavior might be helpful as communication will be much faster in case the cache has some useful data present in it. Also, mapping should be created such that the cache pollution should be avoided if possible. Using FIFO scheduler instead of round robin whenever possible is one solution to avoid cache pollution.

When considered the communication between the hosts, the amount of time is assumed to be the same across any two hosts which may not be correct for all the hosts. So communication delay depending on host pairs should be considered. Also considering the collision due to simultaneous accessing of the shared Ethernet cable by two or more tasks and Ethernet setup time will make the simulation results more accurate and realistic.

There has been some research on cluster schedulers which will schedule tasks across multiple hosts which may prove helpful and efficient [24]. Also, considered approach leaves the gaps in the schedule. This can be solved using tasks splitting where a task can be split in to multiple tasks, preserving the dependency such
that it will fill the gaps in the generated schedule. These techniques can further improve the throughput as it can take advantage of the gaps present in the schedule.
References


[20] Intel Server Board S2600C0 Family: Technical product specification


Appendix I Throughput Viewer

ASML has developed a tool called Throughput Viewer (TPTview) for throughput analysis. TPTview provides the following functionality [14].

- Import and combine different kinds of TWINSCAN traces.
- Visualize sequences in Gantt Charts and allowing an understanding of how the system works.
- Measure timing in between actions, do statistics and verify with budget.

There are various kinds of traces that can be imported and analyzed in TPTview. The most relevant tracing for this thesis is throughput tracing (TP event tracing). Throughput tracing represents the task execution in terms of CPU on and CPU off time. As this thesis is dealing with tasks executing on the main host, file used for analysis is "tpt_short.TP0_cpu". This file contains detailed execution information of the main host tasks. To generate this file, the Computer System Execution Tracing (CSET) tool has to be enabled. CSET internally uses DTrace scripts which can only be executed on Solaris systems. Figure 1 shows a snapshot of TPTview. Information displayed is per core.

Although, TPTview is useful in finding the actual execution times and the communication time between the processes, there is no information about the dependencies between the task. To verify that actual dependencies between the tasks do exist, the task code and the documents regarding the task flow are analyzed. On the basis of the data analyzed, it was found that the tasks can be represented by directed acyclic graphs.

![Figure 1: TPTview](image)
Appendix II Application model Implementation

Figure 2: Screen shot of input csv file

The application model is provided to the task mapping algorithm by representing the DAG in csv format. Figure 2 shows the ‘csv’ file representation of task set III shown in Figure 8.17. The second column always represents the execution time of the task in micro-seconds. The remaining columns are used to convey the dependency information to the algorithm. The data dependency is specified in terms of “kilo-Bytes”. The fixed communication time can also be specified in terms of microseconds if required. A fixed communication time is discriminated from data dependency using decimal symbol, so if fixed communication time of 40 milli-seconds is present between two tasks, it is represented in “csv” file as “40000.0”. The highlighted cell in Figure 2, suggests that ‘task8’ depends on ‘task3’ and data of approximately 1.28 GB of data is communicated between these tasks.

The first step towards generating the mapping is to find the critical path. Input ‘csv’ file is read and tasks and their information is stored in linked-list data structure. Task 0 is added as start task with negligible execution time. Then the calculations related to finding the ideal release time, critical predecessor are done and information is again stored in the form of linked-list.
Appendix III Simulation results for some more task sets

Taskset IV

This task set is chosen to show the limitation of just using the 'Modified Critical Path with capacity' algorithm

![Taskset IV Diagram](image)

Figure 3: Taskset IV

This task set is mapped on 2 hosts having 4 cores per host. When mapped using just 'Modified Critical Path with capacity' it gives mapping as shown in
Figure 4. As seen the critical path is not satisfied even though there is enough capacity. Task 9 execution time exceeds the critical path time. This is because task3, task6, and task7 share the core causing delay in finishing of task3 and starting task 9.

The simulation with MMCP Algorithm, used in this thesis is shown in Figure 5. As seen mapping generated by the MMCP just finishes within the critical path time and Task3, task6, and task7 all are mapped on different cores.

Limitation of host mapping algorithm

Consider the task set IV again shown in Figure 3. As seen from Figure 5, it can finish the execution of complete task set within critical path time if the platform has two hosts each with four cores each, so overall eight cores. Now if these eight cores are distributed as four hosts having two cores per host then communication becomes bottleneck for such a configuration and task set is not finished within critical path time. This is shown in Figure
Taskset V

This task set shows the behaviour of the Algorithm under fork and join type of DAG.

The POOSL model simulation of task set V over two hosts, with two cores per host is shown in Figure 8. As seen core 3 and core 4 (host 2 core -1 and core-2) are used only when necessary to reduce communication over the Ethernet.
Nodes of the tasks in the critical path are indicated in **bold**

Figure 7: Taskset V
Figure 8: Simulation of Taskset V

Tasks running on the same core are represented by the same color.
**Taskset VI**

Taskset VI illustrates the behaviour of the mapping algorithm when more than one independent paths are present in the application model.

![Figure 9: Taskset VI](image)

This task set is executed on two hosts having two cores each.

![Figure 10: Simulation of Taskset VI](image)
Taskset VII

This task set shows the improvement achieved by MMCP over the original MCP algorithm.

Consider the task set shown in Figure 11. Task 6 has highest ALAP release time so it gets scheduled last. If original MCP is used then the mapping generated is shown in Figure 12. As MCP tries to find the core where task can start executing at earliest, it maps task6 with task3 causing delay in starting of task 3 which in turn causes the task graph to overshoot critical path time. The mapping generated by MMCP is shown in Figure 13

Figure 11: Taskset-VII

Figure 12: Mapping generated by original MCP for task set VII

Figure 13: Mapping generated by MMCP for task set VII
Appendix IV Description of Application and Platform model and its Generation Algorithm

In POOSL, various types of classes can be specified to model the system hierarchically. In Figure 6.1, Application and MPSoC represent the cluster classes which can contain various other process classes. SimulationController and Tpview are process classes. Process class defines the behavior of the class using methods written in POOSL. These methods can contain various data classes which are data types of the variables/constants used in methods.

Application Model

Application model represents the structure of the application. It will contain the tasks present in the application and their dependencies. Figure 14, shows the structure of sample POOSL application model constructed using the SHEsim. As seen in Figure 14, the SHEsim displays some extra information about the model along with the graphical view of the model. The graphical interface is used to model the dependencies between the tasks. Each task will be specified using the POOSL. The SHEsim also displays the various interfaces. The Port Interface shows the ports that are available for the application model to communicate with the other models. Message interface specifies what messages are transferred over the available ports along with the data types of message transferred. Information in the Port Interface and the Message Interface is automatically generated from POOSL model of the task and graphical Model. Instantiation Parameters are the parameters which user has to specify. It contains information like mapping of the tasks, their priorities etc.

Figure 14: SHEsim Application Model

Figure 15, shows the browser to specify the task using POOSL. As seen, most of the sub-windows are same as the application model. But now instead of graphical interface, there is option for Edit Method. This is used to specify the various POOSL methods which will define the behavior of the task. Also, the port interface and the message interface will contain information about the...
ports and messages used by the particular task. There is information about the Initial method of the task and the super-class of the task.

All the tasks modeled for this thesis, follow a same sequence of method calls. In the initial method there are calls to two other methods which are “Notify-BuffersAboutMapping” and “NotifyPlatformAboutMapping”. These methods will notify the buffers and platforms about the mappings of the tasks on cores specified in platform model. Then Task waits in “Fire” method till all the dependencies of the task are satisfied and it can start executing. If all the dependencies are satisfied, task will be made ready for execution and when actual execution starts, the space required by the output of the task is reserved in output buffer and the data is written to this buffer at the end of execution.

Figure 15: SHEsim model of single Task

Platform Model

Platform model represents the hardware which will run the task/application model. A cluster class called platform is created which again contains the cluster class called Node. Number of Nodes are equal to the total number of cores present in the system. Each core will have host name affiliated to it. The number of cores assigned per host depends of the cores per host in the considered platform model. Each Node will have the process classes called “Operating System”, “Processor”, “Buffer Memory”, “Data Memory” and “Interconnect”. Out of these, “Operating System” and “Processor” class are most important as they define how tasks from application model will be scheduled on the processor core.

Algorithm for creating POOSL model automatically

This subsection explains how the POOSL model is generated automatically for changing application and platform model. The POOSL is object-oriented language and it has some classes (process, data cluster) as explained in Chapter 6. Many classes do not change with the change in the application model or the platform model. So these classes are saved to one file and these classes are copied to the “POOSL model”. For iterating the task sets continuously till the end of simulation time, two extra tasks are added which will mark the start and end
of the task set and serve as single source and single sink for entire DAG. These tasks have negligible execution time. Then cluster class application specifying the desired throughput, which is calculated from critical path time of the DAG, is added to the model. For controlling the execution of tasks, control buffer are added from the start task to every other tasks. A data buffer is added between the tasks which communicate. Then the behaviour specifications of cluster class Platform are added which will be depending on the overall core count of the platform model under consideration. As the system is considered to be homogeneous, all the cores will have similar specifications like processing power, power consumption, voltage scaling etc. At last, the task classes are added which extends the class called “Task”. New class is created for each task. Depending on the data dependency, following methods are added to the tasks class

1. CheckTokenAvailability: Check if the preceding task/s has written the data to buffer

2. Execute: This method is inherited from the class “Task”.

3. ReleaseSpaceForReads: Releases the reserved space in preceding data buffer after data is read by the task.

4. ReserveSpaceForWrites: Reserves the space for writing in subsequent data buffer/s

5. PerformWrites: Writes the data to the reserved space at the end of execution.

![Algorithm to create POOSL model](image)

Figure 16: Algorithm to create POOSL model

This Algorithm is shown in Figure 16.
Appendix V Description of Tools used

Ftrace

Ftrace is an internal kernel tracer designed to find out what is going on inside the kernel [6]. There are many different tracers available that can be configured. Tracer “sched_switch” is selected for the current application as it traces the context switches and wake ups between tasks. A snapshot of tracing file generated by “sched_switch” is shown in Figure 17. As shown in the Figure 17, sched_switch tracer provides all the useful information like task name, pid, core name it is executing on and timestamp since the system was last rebooted. The function column is explained below. “R, D and S” represent the task states. They are used to represent task states “Running (on run-queue)”, “Uninterrupted sleep” and “Sleep” respectively. The sign “+” indicates addition of new task to the run-queue. The sign “==>” indicates giving up the core to another task. So we know the exact time when process is on the core and when it is off the core.

Grasp

Grasp is a versatile trace visualization toolset which can read ftrace files and represent the execution in graphical format[12]. A screenshot of Grasp window for execution of task set according to mapping generated by algorithm and execution for mapping generated by Linux scheduler is shown in Figure 8.15 and Figure 8.16 respectively.

GTKWave

GTKWave is a fully featured GTK+ based waveform viewer which reads FST, LXT, LXT2, VZT, and GHW files as well as standard Verilog VCD/EVCD files and allows their viewing. GTKWave is developed for Linux, with ports for various other operating systems [13]. Trace files are first converted into VCD format using a plug-in and then visualized using GTKWave.