A Master Thesis presented to

Delft University of Technology

Faculty of Computer Engineering

Techniques for Memory Mapping on Multi-Core Automotive Embedded Systems

For the Degree

Master of Science (M.Sc.) Embedded Systems

By **Rakshith Amarnath**

Master Thesis Number: CE-MS-2012-07

Date of commencement: 01.10.2011

Date of submission: 08.06.2012

Abstract

The demand to increase performance while conserving power has led to the invention of multi-core systems. The software until now had the convenience of gaining better performance over faster processors without any need for a change. The advances in the multi-core hardware have shifted the responsibility of software performance from hardware architects to software developers. To harness the true power of the multi-core, the software must utilize the available cores and memories. In this thesis, we focus on the issue of multi-core memory mapping which is an area of active research.

The availability of multiple memories creates several possibilities for memory mapping. Further, with an increase in the number of memories and application parameters there is an exponential increase in the number of possible mappings. In this context, the challenge is to find techniques which automate the process of finding an efficient mapping for a given use case. The use case under consideration is an automotive software running on a multi-core electronic control unit (ECU). The proposed techniques help to optimize memory accesses by performing efficient memory mapping and reduce the runtime on a system which employs a non uniform memory access (NUMA) characteristic.

The optimal memory mapping problem is NP-complete, it is tackled using integer linear programming (ILP) for smaller problems and heuristics to practically solve bigger problems. We also propose metaheuristics as an add-on to mitigate the drawbacks of ILP and heuristics. The experiments on the dual-core ECU hardware show that our flash memory mapping techniques reduce runtime by 2.76% when caches are enabled and up to 8.73% when caches are disabled. Also, the benefit of the ILP technique for RAM is 50.48% when compared to the placement of all the variables in global RAM.

Acknowledgments

"No Man is an Island" and there are many people whom I would like to thank. First and foremost, I would like to thank my parents and well-wishers who supported me during crucial times when I wanted to pursue my higher studies abroad.

I am very grateful to my supervisors Mr. Simon Kramer, Dr. Zaid Al-Ars and my energetic manager Dr. Jochen Haerdtlein. I thoroughly enjoyed the brainstorming sessions we had and you always helped me in thinking better. My research was carefully supervised by Simon and I thank him very much for his advise, patience and support. I thank professor Zaid for his cheerful cooperation, guidance and help with the administrative formalities. My good friend Soeren Braunstein helped me in various phases of the thesis both professionally and emotionally. Thank you Soeren. I thank all my reviewers who took painstaking efforts in identifying the preliminary errors in my report and also my multi-core team members who gave valuable inputs during our team meetings.

I express my gratitude to the Bosch organization for giving me this wonderful opportunity to perform research. Sincere thanks to Delft University of Technology, its motivating faculty and the Erasmus study abroad program. I am lucky to have good friends who made my stay in Germany and Netherlands a memorable one and I also thank them wholeheartedly.

Contents

1 Introduction **1**

1.1 Research Incentives 2
1.2 Problem Statement and Contributions 2
1.3 Thesis Outline 4

2 Background **5**

2.1 Trends in Automotive 5
2.2 Multi-Core 6
2.2.1 Trends 6
2.2.2 Overview of Architectures 7
2.2.3 Multi-Cores in Automotive 8
2.3 Related Work 9
2.4 Summary 13

3 Techniques for Memory Mapping **14**

3.1 The Memory Mapping Process 15
3.1.1 Terminology 15
3.1.2 Memory Mapping 16
3.1.3 Memory Mapping Strategies 17
3.1.4 Steps in Memory Mapping 19
3.1.5 Mapping of Stack Data 19
3.2 ILP Formulation for Memory Mapping 23
3.2.1 Notations 23
3.2.2 Formulation 24
3.2.3 Assumptions 26
3.2.4 Drawbacks of the ILP 27

3.3 Greedy Heuristics for Memory Mapping . . . 27
3.3.1 Greedy Heuristic for the Placement of Mapping Parameters . . 28
3.3.2 Assumptions . . . 30
3.4 Practical Considerations . . . 30
3.5 Summary . . . 32

4 Memory Mapping for an ECU **33**
4.1 Environment and Setup . . . 33
4.1.1 Application . . . 33
4.1.2 Workstation . . . 34
4.1.3 Target Hardware . . . 35
4.1.4 Settings for Theoretical Evaluation . . . 36
4.2 Theoretical Evaluation . . . 36
4.2.1 ILP versus Greedy Heuristics . . . 36
4.2.2 Selecting an Algorithm . . . 44
4.2.3 Algorithm versus Naive Mapping . . . 45
4.3 Evaluation on a Dual-Core ECU Platform . . . 46
4.3.1 The Effect of Distributing Parameters in Flash . . . 47
4.3.2 The Effect of Distributing Parameters in RAM . . . 50
4.4 Summary . . . 51

5 Refinement using Metaheuristics **53**
5.1 An Introduction to Contention . . . 53
5.2 Limitations of ILP and Greedy Techniques . . . 54
5.3 Metaheuristic I . . . 55
5.3.1 Operating Principle . . . 55
5.3.2 Design . . . 55
5.3.3 Assumptions . . . 57
5.4 Metaheuristic II . . . 59
5.4.1 Operating Principle . . . 59
5.4.2 Design . . . 59
5.4.3 Assumptions . . . 60
5.5 Evaluation . . . 60
5.6 Summary . . . 61

6 Conclusions and Recommendations **62**
6.1 Conclusions . . . 62

6.2 Recommendations 63

G Appendix **66**

G.1 Main Working Topics of the AUTOSAR 66

G.2 AUTOSAR Layered Software Architecture 67

1

Introduction

A modern passenger car hosts several embedded systems and offers a wide range of vehicle safety and comfort features such as anti-lock braking system (ABS), electronic stability control (ESC), engine control, and adaptive cruise control (ACC) among others. These features are provided by electronic control units (ECUs) alongside sensors and actuators. The term ECU refers to microcontroller hardware plus the real-time automotive software. In contrast with traditional mechanically controlled systems, modern drive by wire systems are software controlled. The automotive software runs entirely on a single core in the current generation ECUs. Due to the increasing demand for performance and features, the embedded market has turned towards multi-core technology for efficacy. The paradigm shift brought by the multi-core revolution has changed the way we manage and run the software. The multiple cores and memories create numerous possibilities in their usage. The challenge lies in the selection of the best solution from a large set of solutions. Thus, efficient software techniques are needed to utilize hardware resources of multi-core. In this thesis we focus on techniques to solve the challenges that arise in effectively utilizing the memories of a multi-core embedded system. Though the automotive use case is the target, the concepts can be suitably adapted to other fields with similar challenges. This chapter highlights the research incentives, problem statement and the thesis outline.

1.1 Research Incentives

Modern ECUs running intensive software applications demand high performance from the hardware. In a uniprocessor, the performance improvement was achieved by increasing the clock frequency. This trend could no longer be continued because of diminishing returns in performance, mainly due to: the difficulties in exploiting further benefits from instruction level parallelism, increasing gap between processor and memory speeds, and abnormal increase in power dissipation with increasing frequency. An answer to these technological challenges is the multi-core architecture. The term *multi-core* refers to the processing technology which integrates multiple processor cores on a single die of silicon to provide improved performance at lower power.

Since the introduction of multi-core hardware, the burden of software performance has shifted from hardware architects to software developers [25], and this is driving research towards development of techniques to parallelize the application which efficiently utilize the available cores and memories. Further, the trends in automotive standardization such as AUTomotive Open System Architecture (AUTOSAR) also help to identify inherent parallelism by organizing the software architecture into independent layers. Reducing the dependency between software modules allows for an effective usage of multi-cores. These incentives and trends are driving the automotive industry to switch towards multi-core.

1.2 Problem Statement and Contributions

Software design for multi-core is extensively studied in the automotive industry. In the central area of research and development at Robert Bosch GmbH, our team specializes in the development of software methods that assist the migration of legacy automotive software towards multi-core.

To efficiently utilize the cores and memories, the tasks [1] (work-loads) have to be distributed and the memory has to be allocated. These activities are called task distribution and memory mapping respectively and there is no rule of thumb dictating the order of performing them. It is important to note that both the activities are interlinked i.e. the location where the memory is allocated for a task depends on

[1] A task is the smallest schedulable unit managed by the OS

the core to which the task is assigned and vice versa. However, performing task distribution and memory mapping together results in a problem of bigger complexity. Therefore, in our research group, we have decided to distribute tasks first and then map the memory. Task distribution is already studied and thus the central focus of this thesis is memory mapping on multi-core ECUs.

The benefit from task distribution is partial because the memory now forms the bottleneck for further performance benefits when all the cores access the same memory. The bottleneck increases the memory access time of the application and also leads to indeterministic task response times which cannot be tolerated in safety critical embedded systems where predictability is also a requirement. In addition, the importance of memory mapping is high when the memory access time is a significant fraction of the total program execution time. This is true especially for software intensive automotive applications which frequently access memory. Therefore, after the tasks are distributed, distributing the memory allocation forms the next crucial step towards performance optimization in multi-core.

Memory mapping can be done in several ways. A random mapping creates sub-par application performance especially in architectures where the memory accesses are not uniform. Such architectures are termed as non uniform memory access (NUMA) where the distance of a memory from the core determines its access time and thus local memories can be accessed faster than remote ones. Therefore, on these architectures, it becomes important to leverage the data access locality and choose optimum placement. But optimal memory placement is guaranteed when all the application parameters are examined for their placement in different memories before choosing the best placement. One approach is to manually examine all possibilities to find out the best placement which becomes laborious and impractical when either the number of application parameters or the number of memories change. The other approach is to avoid rework by automating the memory mapping process by using algorithms. The challenge lies in providing efficient techniques that scale well with the underlying architecture.

Multi-core machines featuring several cores and memories inevitably use the NUMA architecture [24, 15]. Lack of memory mapping or sub-optimal memory layout creates severe bottlenecks in memory accesses and hinders performance. It is desirable to distribute the memory accesses in such a way that locality of accesses are preserved and remote accesses are reduced. Therefore it is important for task distribution to be backed up by memory mapping. This brings us to the main problem statement of

this thesis: **"To design scalable techniques which perform efficient memory mapping on multi-core systems"** where we hypothesize that: efficient memory mapping techniques help to reduce the total memory access time of a parallelized application which runs on a multi-core NUMA machine.

Optimal memory placement is a problem of NP-complete complexity [23, 21]. With an increase in problem size (number of memories or number of parameters to be mapped), there is an exponential increase in complexity making optimal solutions impractical. For the memory mapping problem on NUMA machines, our contributions are as follows: we use Integer Linear Programming (ILP) to provide optimal mapping for smaller problems and use heuristics (greedy heuristics) when ILPs are intractable. We apply these techniques to an automotive application and show the benefits by performing measurements on the ECU hardware. In addition, we improve on ILP and heuristic techniques to provide metaheuristics as an add on.

1.3 Thesis Outline

In this chapter, we have highlighted the research incentives and the problem statement. The rest of the thesis is structured as follows: Chapter 2 gives the background where the trends in automotive and multi-core are discussed along with the existing techniques in the field of NUMA memory mapping. In Chapter 3 the techniques of ILP and greedy heuristics for memory mapping are presented. Chapter 4 describes the experiments that indicate the benefits of using our memory mapping techniques on an ECU. In Chapter 5, we refine the earlier techniques by proposing meta-heuristics and in Chapter 6 we end the thesis with the conclusions and recommendations for future work.

2

Background

This chapter describes the trends in the automotive domain, brief background about multi-core and existing research related to the memory mapping problem.

2.1 Trends in Automotive

The increasing complexity of software in automotive systems has given rise to standardized software architectures. This is one of the primordial reasons for the genesis of AUTOSAR. The AUTomotive Open System Architecture (AUTOSAR) provides an open and standardized automotive software architecture which is jointly developed by automobile manufacturers, suppliers and tool developers [4]. Currently the AUTOSAR consortium has organizations like BMW, Bosch Group etc as its core partners and several other members. AUTOSAR is responsible for setting the standards which serves as a platform on which automotive applications can be developed.

"Cooperate on standards, compete on implementation" is the simple and powerful idea behind AUTOSAR. Thus AUTOSAR compliant software aims towards reducing complexity of integration while improving the flexibility, quality and reliability. The benefits of the AUTOSAR is not only restricted to original equipment manufacturers (OEMs) but also to suppliers and tool providers. The OEMs benefit by enhanced design flexibility, reuse of software modules across variants, simplified integration and

reduced development costs. The supplier benefits from reduction of version proliferation and the ease of functional development. And finally the tool providers can now develop seamless and optimized landscapes for tools. Thus, AUTOSAR allows for a smoother integration, exchangeability between supplier's solution and manufacturer's applications while also allowing exchangeability between vehicle platforms. Further information related to the main working topics of AUTOSAR and its layered software architecture is included in the Appendix G.

2.2 Multi-Core

As seen in research incentives there is a demand for better performance with lower power consumption. The only solution for such requirement is the shift towards multi-cores and this section we highlight the multi-core trends, architectures and their usage in automotive industry.

2.2.1 Trends

In processors with a single core, the increase in performance was brought by increasing the most important factor, the clock frequency. Software development on these machines was under the assumption that every processor generation would run much faster than its predecessor [25]. This era of steady growth of single processor performance is over due to the practical limits on power dissipation and the lack of exploitable instruction-level parallelism. Figure 2.1 shows different techniques to increase processor performance using higher clock frequencies or more transistor gates (circuitry). It is seen that multi-core has better performance per milliwatt. Therefore, the common practice to increase processor performance is by placing multiple computational cores on one die, the *multi-core* architecture. Multi-core was initially targeted towards desktop computing but new devices for embedded applications are increasingly adopting multi-core architectures [16]. In future, it is expected that multi-core chips will be readily available at a cheaper price due to large-scale manufacturing by all device vendors.

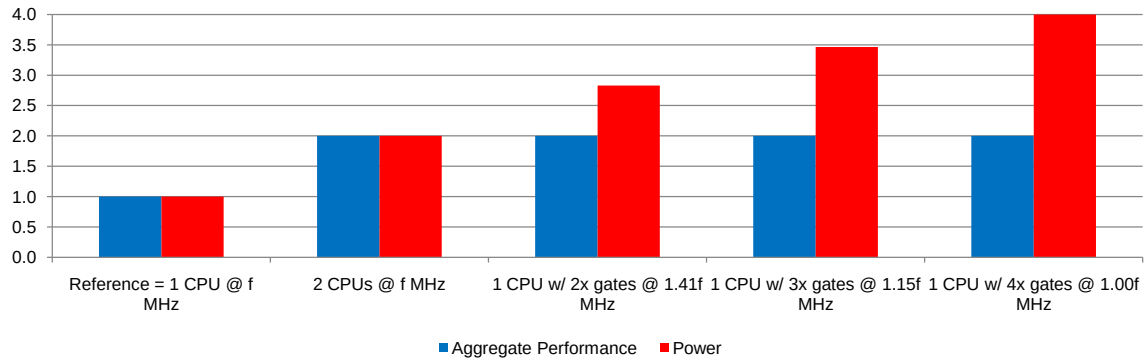

Figure 2.1: This figure shows the alternatives to improve performance. It is seen that by increasing the clock frequency or increasing the number of gates (transistors), power dissipation problems overshadow the performance benefit [16].

2.2.2 Overview of Architectures

Multiprocessors differ from multi-core architectures where the latter implements the features of former in a single package or chip (a die of silicon). For this reason, multi-core is also known as chip-level multi-processor (CMP). As per Flynn's taxonomy, a multi-core machine is classified as a multiple instruction, multiple data (MIMD) [1] processor. The interconnect is the medium responsible for the connection between cores. Depending on the cores used, we can distinguinsh two types of multi-core architecture i.e. homogeneous and heterogeneous. In homogeneous multi-core, the cores have the similar processor and instruction-set architecture for example - Intel Core 2 Quad Processors [19]. Heterogeneous multi-core on the other hand contains at least one core that has a different or customized architecture for example - the Cell Broadband Engine [12]. Of late, there is a new trend where a system employing several tens of cores are regarded as *many-core*. We do not distinguish them in this thesis and regard both types as multi-cores hereafter.

The memory subsystem in a multi-core architecture can be configured in several ways. Based on how the memory is accessed, they can be classified either as uniform memory access (UMA) or as non-uniform memory access (NUMA) machines. The differences between these multi-core memory architectures are shown in Figure 2.2. In UMA architectures, memory accesses are uniform or symmetric i.e. all cores require same

[1] Multiple autonomous processors simultaneously execute different instructions on different data.

time to access memories. For example, central shared memory equally accessible by all the cores. In NUMA machines, the accesses to the memories are not uniform i.e. some accesses are faster than the others. This difference is due to the fact that memory access time depends on the distance of a memory from the processor. In such case, the cores have faster access to local memories than to remote memories. It is the presence of NUMA architecture which makes memory mapping an important step and the ignorance of NUMA can result in sub-par application memory performance [20].

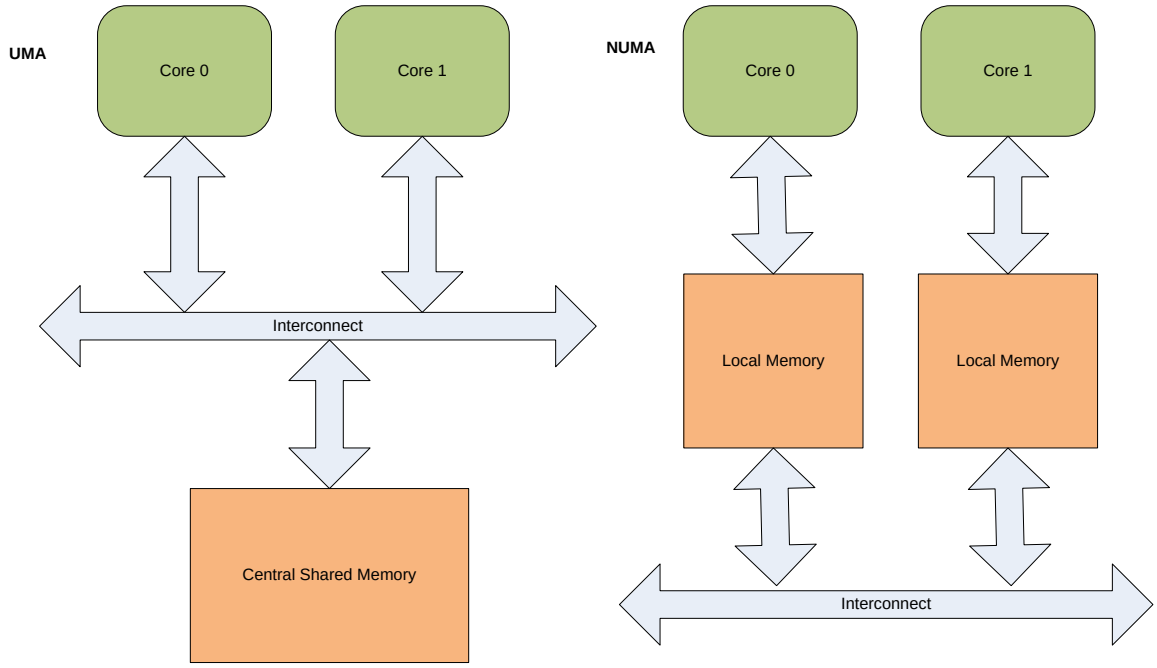

Figure 2.2: In UMA, accesses from cores to central shared memory is uniform. In NUMA, a core can access its local memory directly via a fast bus but the requests to the local memory of other cores have to be routed through the interconnect. This leads to non uniform memory access.

2.2.3 Multi-Cores in Automotive

The automotive industry uses a customized microcontroller multi-core architecture. Since predictability is a key requirement for these real-time systems, the hardware varies widely in comparison to general purpose multi-core machines. The software is static in nature i.e. the schedule sequence and priorities are fixed and there is no need

for hardware supported features such as memory management and virtual memory. In some cases, though there are hardware features like branch prediction and caches to support better average case performance, their usage is very restricted as it hinders determinism.

A UMA machine does not scale well with cores due to the bottleneck in accessing memory. Keeping the future scalability into account, microcontrollers in the automotive domain are expected to have NUMA architectures. In some cases vendors adopt a hybrid approach where they offer symmetric access to some and asymmetric accesses to other memories. The Infineon TriCore system is an example and from [18], it is seen that the local memory of a core forms the level 1 (L1) of the hierarchy and has the fastest access from that core. Remote memory (memory not local to the core) forms the level 2 (L2) of the hierarchy and thus take longer time to access.

AUTOSAR has specifications for multi-core hardware and memory. The hardware assumption by the AUTOSAR indicates that cores should have identical Instruction Set Architecture (ISA) and identical data representation. The memory is accessible to all the cores using a single address space. These guidelines dictate the design of multi-core hardware in the future.

Though there can be many customized hardware implementations, our hardware primarily uses flash as a storage medium for read only data and Static Random Access Memory (SRAM) to store the read write data. Since the memory capacity requirements of our systems is small, the hardware hosts all the memories on the chip and does not use off-chip or external memory interfaces, though this possibility is offered. The techniques proposed in this thesis can also be extended to other architectural configuration.

2.3 Related Work

In modern multi-core machines which scale to a large number of processors, non uniformity of memory access has become an inevitable feature of memory architecture thus giving rise to NUMA. One implication of the NUMA architectural design decision is that the placement and movement of code and data become crucial to performance [24]. Therefore “Data Locality” dictates the performance on NUMA machines. This problem is a topic of extensive research and this section highlights some of the related work referred during literature analysis.

The problem of placing data between on-chip and off-chip memory is studied by Panda et al. in [30] where a strategy to partition variables of an application code is presented. The goal is to minimize the total execution time of embedded applications by techniques which efficiently make use of on-chip scratch-pad RAM (SPRAM). Memory on the chip refers to either cache or SPRAM. Though cache and SPRAMs are in most cases made of static random access memory (SRAM), their functionality is contrasting. Caches flush in and flush out data resulting in an uncertainty (hit or miss) of finding the data in cache. This is done to exploit locality of reference in order to improve the average memory access time. On the other hand, data used in SPRAMs are guaranteed to be present during the lifetime of the program and there is no uncertainty. At first, the decision regarding placement of variable on or off the chip is analyzed. Variables placed on the chip are further examined for conflict misses in cache placement. Conflicting variables are identified and mapped to SPRAMs. Then, variables possessing high locality of reference and fewer conflicts are placed in cache. From benchmark experiments, they show that efficient distribution of memory between cache and SPRAM resulted in improved performance. The work done by Panda et al. in [30] and [29] gives a good starting point to understand the benefits of optimum memory placement in embedded systems and memory exploration strategies. In this thesis, memory exploration is not possible as we use packaged (off-the-shelf) processors. Thus keeping NUMA architecture and memory placement as the focus, we refer to other related works.

Avissar et al. in [8] propose an efficient compiler method by modeling the problem as a 0/1 ILP to distribute data among several memory units in an embedded processor. They target embedded systems with scratchpad memories without caches. The ILP formulated has the objective of minimizing the total memory access time of all the memory accesses in the application. The work done by Avissar et al. has majorly influenced the ILP formulation of this thesis. This formulation is suitably adapted to handle multi-core use case and is discussed in subsequent chapters. It is claimed in the paper that for large programs with few thousand variables, the ILP is fast. But even a small sized automotive application considered in this thesis takes a huge amount of time and in some case the ILP does not give a solution when the constraints are too strict. These experiments are discussed later. Due to the exponential increase in problem complexity, it is evident that big problems (many-core use case) cannot be solved using ILP and hence we also propose heuristics to give practical solutions to it. Despite a few shortcomings, their work provides a starting point for the ILP formulation in this thesis.

The exploration of on-chip memory architecture for embedded System-on-Chip (SoC) is discussed in [23]. This work deals with exploration of memory architecture and data layout optimization of the explored architecture. We only refer the data layout optimization as it is the main topic of this thesis. The author considers a Digital Signal Processor (DSP) and an on-chip memory architecture which is organized as multiple memory banks. DSPs typically have two or more address generation units and multiple on-chip buses to facilitate multiple memory accesses. It is for this reason, the memories are organized into multiple banks to facilitate simultaneous data accesses. In contrast to DSPs, we have microcontroller units (MCUs) that generate only one memory access per cycle. Therefore the memories of cost effective MCUs seldom have multiple banks. However, the introduction of multi-core now offers the possibility of more than one access (by two different cores). The work done in [23] focuses on data dominated DSP applications rather on control-dominated MCU applications. Their techniques involve an ILP formulation, formulation of a genetic algorithm (GA) and a heuristic for optimizing the data layout. There is a notable difference in the their approaches and this thesis. The author is able to get the number of simultaneous accesses to data sections and uses this as an input to minimize memory access stalls due to conflicting access of data sections placed in the same memory bank. However, the same technique cannot be used in our case for these reasons:

1. Simultaneous accesses depend on task distribution. Profiling for simultaneous accesses is a tedious process and it has to be repeated for every change in task distribution.

2. Different relationships between the use cases. In case of DSP, there is a one to many relationship - i.e. one processor needs different data in one cycle. But in multi-core MCU, there is many to one relationship - i.e. many different cores contend on the memory when they need same data. And problem arises when this data (neglecting duplication) resides on only one memory.

Therefore in this thesis, the attempt to minimize contention is stochastically performed by calculating probabilities of contention. This aspect is discussed later during the design of meta-heuristics.

Automatic optimization in using memory resources of high performance, scalable computing systems has been studied in [14]. The allocation problem is viewed as a combination of optimal packing problem and contention reduction problem. The problem is encoded such that it is manipulated by a GA. The GA starts assuming a random

initial population, then via the process of crossover (60%) and mutation (40%), evolves the population and selects the best candidate while maintaining diverse population at the same time. The work has adopted different encoding mechanisms such as integer encoding and order encoding to solve the memory mapping problem. In their work, we see the application of GAs to solve the problem of memory mapping.

An approach to combat non-uniformity in NUMA machines is to hide it using a memory management system in the operating system kernel [13]. This work envisions a coherent memory abstraction built into the memory management system. Coherent memory is uniformly accessible from all processors in the system. Therefore, the attempt is to ease the programming of NUMA multiprocessors without losing performance. Though the abstraction hides the anomalies of NUMA architecture by increasing programming convenience, it does so using Virtual Memory(VM) and paging approaches which are not practiced in real-time embedded systems. Moreover altering the operating system kernel is not the focus of our thesis. However with the advent of many-core architectures, we foresee a possibility of virtualization, dynamic scheduling and therefore include these related work regarding NUMA machines. Dynamic task and data placement over NUMA architecture has been studied by Broquedis et al. [11]. The work done by Bolosky et al. [9] also relates to optimal page placement strategies for NUMA multiprocessors. Similarly, Larowe et al. in [24] show that there is no single policy which caters all needs in case of NUMA architectures further indicating that solution to NUMA problems require research on architectural issues, compiler assistance, language constructs etc.

Solutions to NUMA problems have been tackled in various ways starting with support from Operating Systems (OS) to compilers. Exploration of OS support for thread and memory placement on NUMA architecture has been studied in [3] using concepts of thread binding and location specific memory allocation. The compiler techniques to distribute data and computation is studied in [27]. This work makes use of graph theory (directed, connected graphs called LCG - Locality Communication Graph) to represent locality and formulates compiler technique as a Mixed Integer Nonlinear Programming (MINLP) on the LCG. The compiler builds the LCG at compile time and then formulates the optimization problem that looks for the decomposition to minimize overhead. In embedded systems, it is common to use tailored off-the-shelf OS and compilers. Altering the internal behaviour of these packaged modules is costly and tedious. But one has the freedom in altering the methodology of usage in solving a problem. We discuss these methodologies and techniques in the subsequent chapter.

The possibility of significant performance benefits by optimal data placement in NUMA machines has been studied by Bolosky et al. in [10]. It shows that problems like false sharing occur when data items accessed by disjoint sets of processors are inadvertently placed on a common page. It proposes techniques such as trace analysis to overcome false sharing. From their tests, it is shown that performance improvement of optimal mapping over a naive approach is around 25% to 50%. This paper reveals the importance of NUMA problem especially when memory access time is a significant fraction of program execution time. Most application intense automotive softwares spend considerable amount of time in memory accesses and from Amdahl's law it is very important to make a design trade-off which favours the common case. Also, the improvements in CPU performance is outstripping the memory performance, forcing future processors to spend a larger fraction of their time waiting for memory. Therefore it becomes important to optimize memory placement on multi-core NUMA machines.

2.4 Summary

This chapter has highlighted the trends in automotive, the background information related to multi-core and existing work in the field of NUMA memory mapping. From the study of literature it is evident that the placement of code and data in NUMA machines crucially impact performance. Moreover, the benefits are higher if memory access time is a significant fraction of program execution time. In subsequent chapters, this problem is thoroughly analyzed, algorithms are implemented and experiments are conducted to show the performance benefits of efficient memory mapping. Our techniques provide scalable solutions to tackle the memory mapping problem and have advantages in automating the process using algorithms.

3

Techniques for Memory Mapping

This part of the thesis covers the techniques to solve the memory mapping problem and describes in detail the design and formulation of algorithms. For small problems, ILP is used to give an optimal mapping. Bigger problems can only be addressed by heuristics because of the NP-completeness.

This chapter is structured as follows: Section 3.1 describes the memory mapping process, where the Subsection 3.1.1 introduces the frequently used terms and conventions, in Subsection 3.1.2, the memory mapping is illustrated to add more clarity to our problem statement. The Subsection 3.1.3 describes the memory mapping strategy and Section 3.1.4 describes the steps in which our strategies are applied. Before the ILP and greedy techniques are applied, the stack data in RAM is mapped by using a simple technique discussed in Subsection 3.1.5. The ILP and greedy techniques to solve the memory mapping problem are presented in Sections 3.2 and 3.3 respectively. Lastly, the practical considerations of the techniques developed in this chapter are described in Section 3.4.

3.1 The Memory Mapping Process

3.1.1 Terminology

Simple and logical abstractions are made to describe and manage the feature-rich automotive software. The following terminologies are used throughout the thesis.

Task. "A *task* is the smallest schedulable unit managed by the OS" [7, p. 79]. A task has an existence only from the standpoint of the Real Time Operating System (RTOS) which has the scheduler to execute different tasks depending on priority.

Process. A *process* is a (void void) function which belongs to a task. There is an one to many relationship between a task and a process.

Service. A *service* represents common utility functions (sometimes in-line). A process makes use of services and there is a many to many relationship between them.

Label. The global read-write RAM data in the software is called a *message*. Messages are used by several processes to share the data among them. For simplicity here after the term *label* is used instead of a message.

Constant. A *constant* represents a calibration parameter or a system constant which is a read only global variable in the program flash. From the application point of view, calibration parameters have significant role in tuning the software for a desired functional output.

Flash Content. A *flash content* refers to code (processes, services) and constant data (constants).

Mapping Parameter. A *mapping parameter* describes the parameter that is mapped to the memory. Thus a mapping parameter can either be a label, constant, process or a service. We use the term *parameter* and mapping parameter interchangeably.

3.1.2 Memory Mapping

Here we clarify the need for memory mapping with some illustrations. A hypothetical system containing a single core, on-chip RAM and flash is shown in the Figure 3.1. This shall serve as an example to explain the problem. Code and data are mapped to Flash 0 and RAM 0 respectively. Tasks in a single-core system run sequentially and there is one access at a time to each memory. Therefore, the scope of optimization is within one memory where proper alignment of code and data is an important factor for performance improvement.

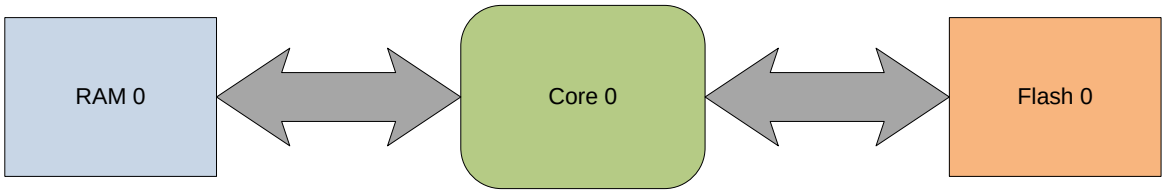

Figure 3.1: Hypothetical single-core system with single RAM and Flash.

The single-core system is now extended to a multi-core NUMA architecture to give a hypothetical dual-core system as shown in Figure 3.2. To utilize the computational capabilities of the cores, the application is parallelized by distributing the tasks to the cores. However the memory mapping is the same as the single-core scenario above. In this case, RAM and flash memory of the other core (RAM 1 and Flash 1) are not utilized. Also the memories of one core (RAM 0 and Flash 0) create a bottleneck due to simultaneous accesses from both the cores (Core 0 and Core 1). This leads to contention [1] which increases memory access time. Even if memory mapping distributes the placement, another reason for increased memory access time can occur due to improper distribution of code and data. For example, data needed by Core 0 takes more time to be accessed from remote memories (RAM 1 or Flash 1) and a poor mapping fails to consider the non-uniformity of memory accesses. Therefore, the objective of memory mapping is to minimize the total memory access time by distributing the memory placements optimally among different memories. Such systems are not merely a hypothesis. Upcoming multi-core architectures [17] provide multiple memory modules with independent read and write interfaces. Such architectural changes in the hardware necessitate the need for efficient memory mapping.

[1] Due to cost reasons, memories have only one read/write access port which support one access at a time

Therefore, the memory mapping problem deals with efficient distribution of code and data to multiple memories to reduce the total memory access time. Since we deal with NUMA architectures, reduction in access time is also due to maximization (minimization) of local accesses (remote accesses). Hence, we see that task distribution utilizes multiple cores and distributing the memory placement utilizes multiple memories. This way of parallelizing the work load and memory accesses is an important step towards software distribution for multi-cores.

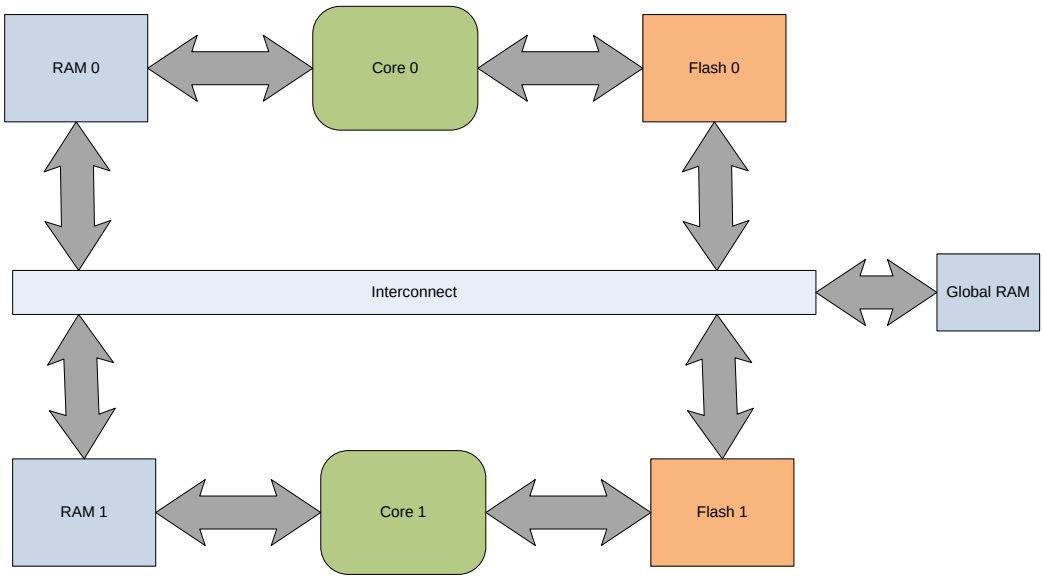

Figure 3.2: Hypothetical dual-core system with multiple RAM and flash memories. Additional Global RAM is added to explain the concepts later. It is assumed that accesses to Global RAM are very costly.

3.1.3 Memory Mapping Strategies

In literature, different (manual and automatic) approaches have been taken to solve the memory mapping problem. But irrespective of the methods used to find the optimal memory mapping, the steps taken to generate the memory layout are fairly the same.

Block	Purpose
Application	This block contains the application code along with the code for the operating system and other drivers
Model	Is the framework on which application and hardware can be represented as inputs to the algorithms. A model generates outputs that assist linking process and helps to theoretically evaluate the algorithms
Model Application	This is a representation of application and profile information in the syntax which is supported by the model
Model Hardware	This block holds the hardware architecture such that the hardware properties are accurately captured
Algorithms for Memory Mapping	This is the block which hosts the algorithms that perform efficient memory mapping
Platform	Hosts the hardware and supports run time measurements via traces

Table 3.1: Blocks of the memory mapping flowchart shown in Figure 3.3

Figure 3.3 depicts the flowchart for the memory mapping. The important blocks are described in the Table 3.1. In the absence of automation, the flow is straightforward and it is necessary to manually optimize the placement of code and data. To avoid tedious human effort in placing code and data in multiple memories, algorithms provide an effective alternative. In the flowchart, the model plays an important role. The model helps in the development of algorithms. Further, the algorithms developed can be theoretically evaluated using the model. The output from the model can be used to assist the linking phase to perform efficient memory mapping and to generate the target specific executable. This executable can be programmed on the hardware to get run time measurements. Another important block is the platform. The platform hosts the hardware, models the vehicle behaviour and supports runtime measurements. The hardware can be a fully packaged off-the-shelf microcontroller or a virtual prototype (soft-core) given by the device vendor whereas the simulation of vehicle behaviour can be achieved through simulation environments like Simulink [26] for example.

3.1.4 Steps in Memory Mapping

Memory mapping strategy varies with the type of mapping parameter and the type of memory to which the parameter is mapped. Therefore, each mapping parameter has a separate algorithm and this section describes the sequence in which these algorithms are used.

Stack is frequently used in high level programming languages such as C. Therefore as a first step, the stack is mapped to RAM. No ILP or greedy algorithms are used to map the stack because the stack is local to tasks in the core and the task distribution determines the stack placement. Algorithms are only used for the distribution of mapping parameters to different memories. The stack mapping is followed by the placement of labels thus completing RAM placement. In case of flash, there is no fixed order to map constants and code. Since constants occupy less space, they are mapped prior to the placement of code. A visual representation of these steps is illustrated in Figure 3.4.

3.1.5 Mapping of Stack Data

Program data consists of labels, constants and stack. Stack is a segment of dynamic memory that holds parameters, local variables and return addresses. Since the performance on a NUMA machine depends on data locality, stack placement must be performed carefully. The methodology to map the stack data is discussed here.

A program stack grows as the control is transferred to a function, while the function exits, the stack shrinks in size. Thus, the stack is reused amongst programs whose lifetimes do not overlap. The operations performed on a stack have the terminology 'push' to indicate placing of data on the stack and 'pop' indicating removal of data from the stack. The stack pointer is responsible to point to the latest entry on the stack. In our case, the stack design is simple. An assumption is made during design time regarding the worst possible size of stack growth on each core. Then the stack of this size is allocated on the RAM local to the core. This is a simple but pessimistic approach because it over-estimates the required memory.

Task distribution gives the assignment of tasks to cores. The stack of the tasks are mapped to the RAM which is local to the core on which the tasks execute. To compute the worst case stack size, it is important to distinguish between cooperative and pree-

mptive tasks. Switching between cooperative tasks occur only at process boundaries. Lower priority cooperative and preemptive tasks can be preempted anywhere by a higher priority preemptive task. When there is a mixture of cooperative and preemptive tasks, the worst case stack size is the sum of stack size of process with maximum size amongst cooperative tasks and sum of worst case stack sizes of all preemptive tasks on the core. It is assumed that this worst case size fits to the memory.

This can be also captured by the following mathematical notation:

Let $W(S_i)$ be the worst possible size that the stack of tasks in core i can grow and M_j the initial capacity of the j^{th} memory. Then assuming that $W(S_i) \leq M_j$, $W(S_i)$ is allocated to memory unit j, where j is the memory local to core i. And let the available space in memory unit j after this placement be $M_{j'}$. Then $M_{j'}$ is as given by Equation 3.1 below:

$$M_{j'} = M_j - W(S_i) \tag{3.1}$$

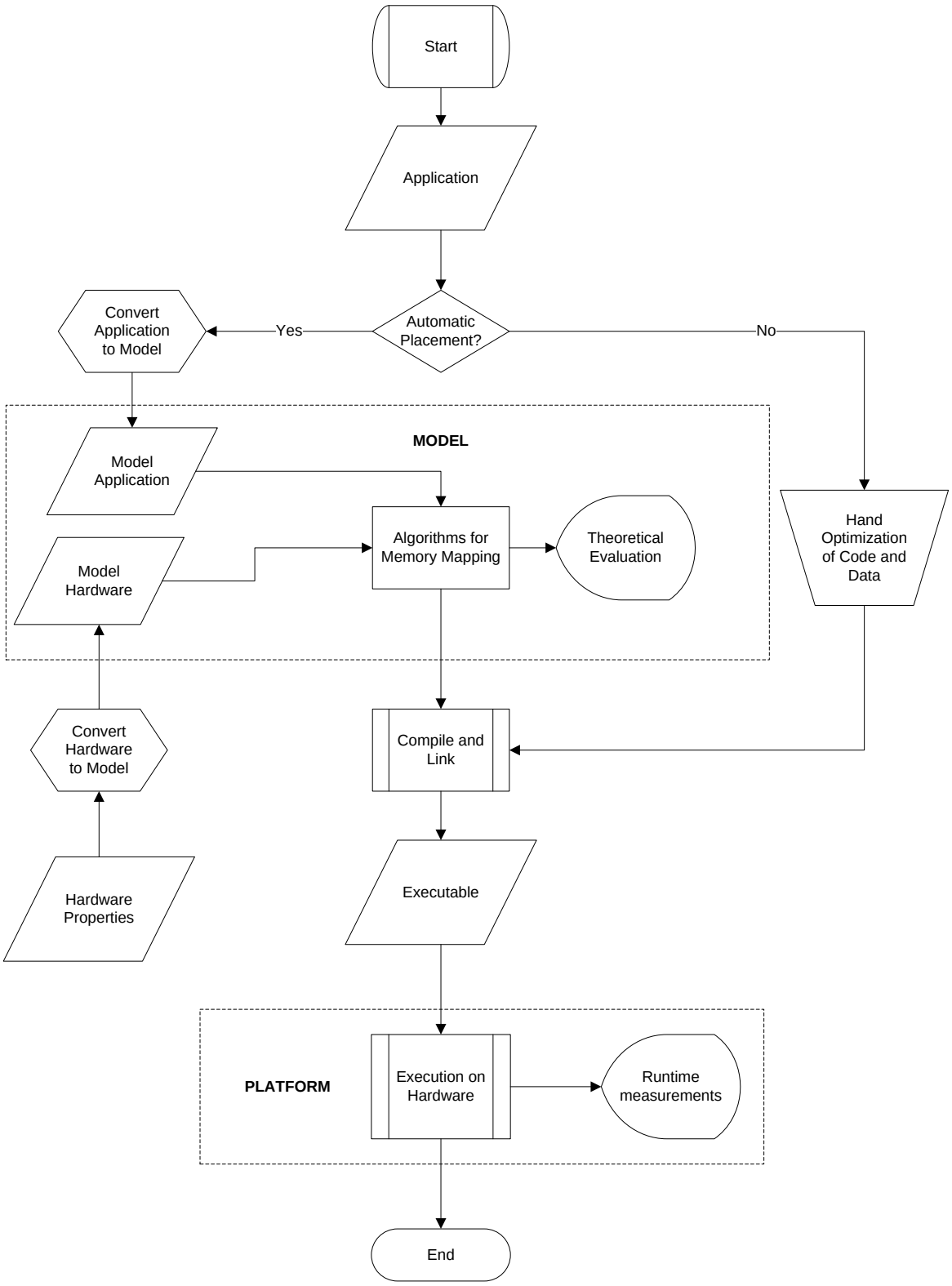

Figure 3.3: This figure shows the flowchart memory mapping. The block 'Algorithms for Memory Mapping' forms the central work of this thesis.

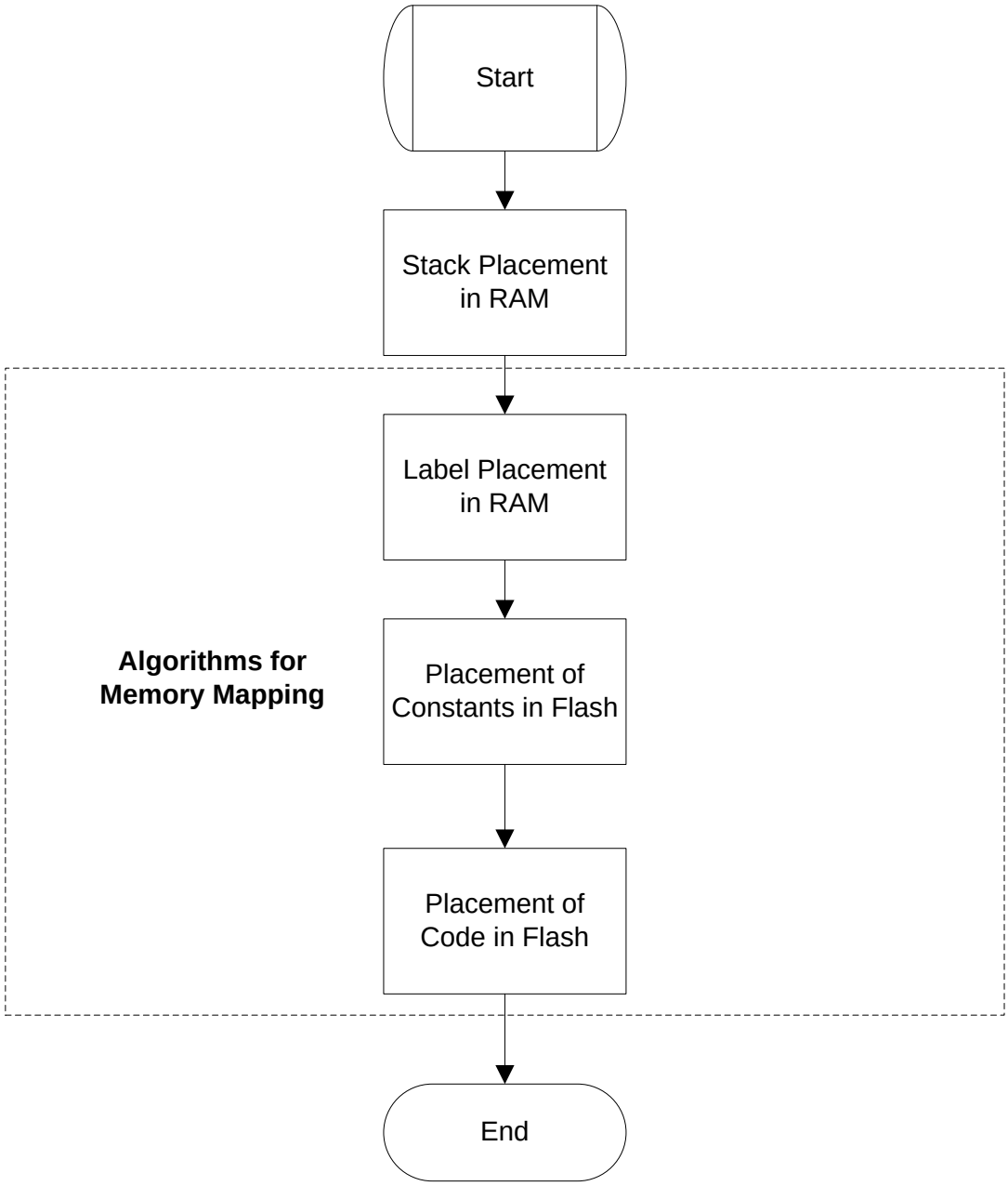

Figure 3.4: This figure shows the sequence followed in applying the memory mapping strategies.

3.2 ILP Formulation for Memory Mapping

Linear programming (LP) is widely used to solve problems in mathematics and operations research where ones objective is to maximize profit or to minimize cost. Integer linear programming (ILP) in particular is a subset of LP problems where the decision variables are integers. And in 0-1 ILP also known as Binary Integer Programming (BIP), the decision variables are either 0 or 1. Such kinds of problem formulations are suitable to optimize digital circuits and other graph theory problems where the outcome is usually a 0 (indicating the absence of something) or a 1 (indicating presence).

The problem of optimal memory mapping is NP-complete [23, 21]. Despite the NP-completeness of the problem, an efficient ILP formulation can be used to get optimal mapping for problems of small dimensions. It can be seen that for a typical multi-core use case there are around two to four cores and three to six memories, the modern ILP solvers are capable of converging to a solution. But, in case of a very large problem size i.e. the many-core use case of around hundred processor cores, many memories and very large volumes of software, there is an exponential increase in problem complexity thus making optimal placement impractical. This motivates us to explore heuristic approaches later in this thesis. However, at this stage it is worthwhile to use an ILP to get an upper limit on the optimality of the solution. Because of its optimality, ILP produces the highest quality solution for small problems. The ILP formulation is inspired by the work done by Avissar et. al [8]. This formulation has been extended and made applicable to the multi-core use case.

3.2.1 Notations

The general notations applicable to mapping parameters are presented here:
Let,

i, where $i \in [1, I]$ be the i^{th} core;

j, where $j \in [1, J]$ be the j^{th} heterogeneous memory;

k, where $k \in [1, K]$ be the k^{th} mapping parameter;

Nr_{ki}, $\forall k \in [1, K]$, $\forall i \in [1, I]$ be the number of times the mapping parameter k is read by core i;

Nw_{ki}, $\forall k \in [1, K]$, $\forall i \in [1, I]$ be the number of times the mapping parameter k is written by core i;

Tr_{ij}, $\forall i \in [1, I]$, $\forall j \in [1, J]$ be the read access times for memory j by core i;

Tw_{ij}, $\forall i \in [1, I]$, $\forall j \in [1, J]$ be the write access times for memory j by core i;
M_j be the size in bytes of j^{th} memory
$M_{j'}$ be the size in bytes of j^{th} memory remaining the after previous placement
S_k be the size in bytes of the k^{th} mapping parameter;
F_k be the multiplicative factor for mapping parameter k
decision variable for placement $P_{kj} = \begin{cases} 1 & \text{if mapping parameter } k \text{ is placed in } j^{th} \text{ memory} \\ 0 & \text{otherwise} \end{cases}$

where I, J and K are the total number of cores, memories and mapping parameters respectively.

3.2.2 Formulation

Objective function

The objective function is to minimize the total access time from all cores to all the mapping parameters in the application. The objective function differs in case of RAM and flash placement and assuming one memory access per cycle, they are given by:

Objective function for RAM - labels

$$Objective\ function_{RAM} = min(\sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{k=1}^{K} P_{kj}[Nr_{ki}Tr_{ij} + Nw_{ki}Tw_{ij}]F_k) \tag{3.2}$$

Objective function for flash - constants and code

$$Objective\ function_{flash} = min(\sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{k=1}^{K} P_{kj}[Nr_{ki}Tr_{ij}]F_k) \tag{3.3}$$

Constraints

The ILP for mapping parameters in memory has the following constraints:

Placement (Equality) Constraint: Every mapping parameter has to be placed in just one memory

$$\sum_{j=1}^{J} P_{kj} = 1, \forall k \in [1, K] \tag{3.4}$$

Capacity (Inequality) Constraint: The size of mapping parameters placed in every memory must not exceed the capacity of the memory

$$\sum_{k=1}^{K} P_{kj} S_k \leq M_{j'}, \forall j \in [1, J] \tag{3.5}$$

where $M_{j'}$ for placement of labels is given by Equation 3.1. In case of flash placement, the constants are mapped first therefore $M_{j'}$ is same as M_j but for code placement, $M_{j'}$ is given in Equation 3.6. Where the second term gives the total space occupied by the constants in memory j.

$$M_{j'} = M_j - \sum_{k=1}^{K} P_{kj} S_k, \forall j \in [1, J] \tag{3.6}$$

It is important to note the reason for the inclusion of F_k in the objective function. This factor takes into account the effect of mapping parameter size on the number of memory accesses. For example, assume that the microcontroller is capable of accessing a word (CPU-word i.e. 32 bits for a 32 bit microcontroller) from RAM in one access. Now labels of size greater than a word need more than one accesses to be fetched from the RAM. Thus, we see that labels of size one to four bytes need one memory access while labels of size five to eight bytes need two memory accesses and so on. Hence generalizing, F_k is given in Equation 3.7.

$$F_k = \lceil \frac{S_k}{Word\ length} \rceil \tag{3.7}$$

In case of flash memory which is accessed at the granularity of a flash line (typically 128 bits), F_k is defined as:

$$F_k = \lceil \frac{S_k}{Flash\ line\ length} \rceil \tag{3.8}$$

In the objective function 3.2, 3.3, P_{kj} is the decision variable which is the result of the ILP solver. The solver gives the optimal placement of mapping parameter k in memory j under placement and capacity constraints such that total memory access time is reduced. Having K parameters and J memories results in $K \cdot J$ decision variables. This shows that the problem explodes into a bigger problem when either the mapping parameters or the available memories increase. In architectures which

support more than one memory accesses per cycle, the objective function does not consider the overlap of memory latencies into account. Also [8] indicates that the objective function in such case becomes non-linear. Hence for the formulation of ILP, it is assumed that there can be at most one memory access per cycle. The ILP can be solved using public domain solvers like lp_solve [1].

3.2.3 Assumptions

For a problem to be formulated as an ILP, the objective function and constraints must obey the rules related to integer linear programming. The assumptions made in the above ILP formulation are highlighted here. These assumptions are related to the access times and number of accesses for each label.

- A core generates at most one memory request per cycle.
- The access times (read access times for flash and read-write access times for RAM) to each memories are known. These values do not change.
- The anomalies introduced by the interconnect and by simultaneous remote accesses are indirectly accounted by considering a NUMA architecture of faster access to local memory and slower access to remote memory.
- Given the task rate, task distribution, the number of reads and writes for a mapping parameter is computed using the call tree [2] of the task.
- Flash memory is a read-only medium and EEPROM (which is an emulation of flash) writing is ignored as it not the common case of performance optimization.

Although the assumptions ease problem formulation, they have their own implications. The assumptions abstract the true behaviour of the hardware and may lead to inaccurate results. At this stage, the assumptions are justified by the reasoning below. The assumption of a single memory access per cycle is because of the MCU use case. Unlike DSP, the MCU generates one memory request per cycle. Further, the memory access times can be taken from the processor documentation. There is a difficulty in accurately modeling the delay introduced by the conflicts in the interconnect. This delay is indirectly addressed when a higher access time to remote memory is considered. Lastly, given a task to core allocation (task distribution) and the process and

[2] call tree gives the number of times the mapping parameters are accessed by a task

service calls of the tasks (call tree), it is possible to statically compute the number of reads or writes to a mapping parameter.

3.2.4 Drawbacks of the ILP

In case of ILP formulation there are many assumptions. The ILP formulation would be more complex in the absence of these assumptions. This ease of formulation has a drawback which impacts the accuracy of the result. Under such assumptions, cost of accessing a mapping parameter from the memory must be calculated precisely without introducing additional inaccuracies between theory and practice.

The assumptions work well when the accesses to memories are not uniform. But for architectures with symmetric accesses, different values of memories have to be deliberately added to ILP to get the distribution. In such cases, theoretical results differ from the practical measurements on the hardware.

The assumption of asymmetric flash access time allows the ILP solver to group the code required by one core to one memory module. But there is a potential problem which may arise. If the cores access the same memory simultaneously then only one of the request gets serviced on time and the others get delayed. This delay is not directly modeled by the ILP but indirectly considered in the assumption of slower access to remote memory. The assumptions are justified at this stage but they are reworked later when metaheuristics are introduced.

3.3 Greedy Heuristics for Memory Mapping

The problem of finding an optimal mapping is NP-complete [23]. The memory mapping problem suffers from the curse of dimensionality where even a small increase in the problem size causes an exponential increase in the complexity. For small dimensions, it is seen from experiments that the ILP is practically fast. But due to the trends in very large scale integration technology, it is likely to imagine large number of cores and memories in the future [2, 22]. For these problems, the ILP technique to find an optimal solution is intractable. This requires us to focus on the design of heuristics which solve big problems quickly and efficiently. Heuristics are subjected to approximations and therefore can only promise near to optimal (in most case), optimal (if lucky) or no solution (even if one exists) at all. There is an uncertainty

with the solution because it does not traverse the entire solution space but makes approximations to traverse the most favoured solutions. It is proven that heuristics are beneficial for most of the problems which suffer from the curse of dimensionality.

In this thesis, we use a greedy strategy to solve the memory mapping problem. This strategy is built into a heuristic which gives an efficient placement of a mapping parameter in its respective memory. A greedy heuristic is an algorithm that makes locally optimal choice at each stage with the hope to find a global optimal solution. A greedy algorithm provides optimal solution to a problem if it obeys the following two properties:

- The greedy choice property - i.e. make the locally optimum solution at every stage and never reconsider the choices made so far.
- The optimal substructure property - i.e. an optimal solution to a problem is obtained by finding an optimal solution to the sub-problems.

The choice of this heuristic is justified as there are affinities in the access patterns of automotive software. This means that the data is frequently accessed by the cores and the benefit is more when the data is placed closer to the core which needs it the most. This is the greedy choice which we hope to exploit at every stage. The design of heuristics and the assumptions are discussed in the following subsections.

3.3.1 Greedy Heuristic for the Placement of Mapping Parameters

This subsection shows the design of a greedy heuristic used in the placement of *Mapping Parameters*. Two algorithms are presented:

- Algorithm 1 - Preparation of the input
- Algorithm 2 - Greedy heuristic to place *Mapping Parameters*.

The initialization needed to setup the greedy heuristic is given in Algorithm 1. The term affinity refers to the total number of reads and writes from a *Core* to a *Mapping Parameter*. The objective of this algorithm is to find the *Core* which has the highest affinity for a given *Mapping Parameter*. This is done for all the *Mapping Parameters* over all the *Cores* to generate a list *CoreParameterAffines* which has the affinities sorted in descending order. The term list is derived from the programming language (Java) used to implement the algorithm. It is used to represent an array in which each element contains the following information: the *Mapping Parameter*, the *Core* which

needs this parameter the most (in terms of reads and writes) and the affinity which is the sum of reads and writes from the *Core* to this *Mapping Parameter*.

Algorithm 1 Preparation of the input

```
for all Mapping Parameters do
  for all Cores do
    CoreParameterAffines ← Add the Mapping Parameter with the most affine
    Core to the list
  end for
end for
Sort the list in decreasing order of their affinity
```

Ensure: *Mapping Parameter* with the maximum affinity is first in the *CoreParameterAffines* list

The greedy heuristic is shown in the Algorithm 2. This algorithm works on the list *CoreParameterAffines* generated by Algorithm 1 to minimize total memory access time by adopting the following greedy methodology:
Until the fastest memory for the mapping parameter is available, the mapping parameter is placed in the fastest (most beneficial) memory of the core. But, when the fastest memory is full, the algorithm recursively finds the next available fastest memory for its placement.

In other words, for every *CoreParameterAffine*, the *FastestMemory* from the core is determined and the *Mapping Parameter* is checked for possible placement in the *FastestMemory*. If the *FastestMemory* is fully occupied then the placement is recursively tried over the next available *FastestMemories*.

Due to similar objectives, the greedy heuristic and the ILP can be compared with respect to two metrics. One is the optimality of the solution and the second is the time taken to solve. It is important to note that the result from the ILP is optimal even under capacity constraints. While, the greedy heuristic can only be optimal under no capacity constraints (i.e. memories are sufficient for the greedy choice to work). From the experimental results discussed in Chapter 4, it can be concluded that the greedy heuristic is very fast in comparison to the ILP and optimal if there is sufficient space in the *FastestMemory* at every stage. Thus there is a trade off, in which the ILP being slow in solving does not compromise on quality while the greedy heuristic compromises on optimality by being very quick. If an application engineer wants to choose one, it depends entirely on the use case (problem dimension) and the tolerance for optimality and solving time. However, it is worth mentioning that

Algorithm 2 Greedy heuristic to place *Mapping Parameters*

Require: *CoreParameterAffines* as computed in Algorithm 1
  for all *CoreParameterAffines* **do**
    for all *Memories* **do**
      Find the *FastestMemory* available from the *Core*
      if *Parameter* fits into the *FastestMemory* **then**
        Map *Parameter* to → *FastestMemory*
      else
        Recursively find the next available *FastestMemory* for the *Mapping Parameter*
      end if
    end for
  end for
Ensure: That insufficient capacities are reported with an error condition
  return In case of no error, return the memory mapping of *Mapping Parameters* to appropriate *Memories*

for problems of large dimensions and problems with stringent constraints, the greedy heuristic outperforms the ILP and it is the only promising approach of the two.

3.3.2 Assumptions

The assumptions made during the design of greedy heuristics are:

- The task distribution, task rate and the call tree of the tasks are the inputs from which the affinities can be computed.
- The access times (read access times for flash and read-write access times for RAM) to each memories are known. These values do not change.
- The memory accesses are non uniform (NUMA) where the local memories are fast and remote memories are slow.

3.4 Practical Considerations

The ILP and greedy algorithms are designed at a fine granularity which makes practical implementation difficult. In other words, the algorithms have considered the

individual placement of mapping parameters in the memory. Although these techniques theoretically give better results as seen in theoretical evaluation experiments later in Chapter 4, in practice a mapping parameter is too small to be mapped to the memory hardware. Therefore, the above algorithms need to be modified by working on a bigger granularity. To achieve this, mapping parameters are grouped to form bigger units and the group of mapping parameters is termed a *section*. Thus a section is a group of identical mapping parameters which can be located in the memory.

The sections can be created in several ways. Compilers provide default sections like .text, .data, .bss to hold the program code, initialized and uninitialized data respectively. New sections can be introduced using compiler and linker specific keywords when there is a need for more customized or controlled memory layout. Also AUTOSAR provides specification for memory mapping [5] which requires every AUTOSAR software module to support different memory types. If the software development is carried out based on the AUTOSAR guidelines, then it would be easier to make use of different sections of each software module. But in the absence of these sections, it is burdensome to change input source files to generate new sections for placement. As an alternative, we make use of the default sections of the object files and locate them on an object file basis in the memory. Section information present in the object files are converted to the equivalent section description in the model. The mapping parameters present in the sections are examined for placement in different memories using the ILP and greedy techniques.

The underlying concepts of the ILP and greedy techniques remain the same. The changed portions are the inputs and the outputs of the algorithms. The inputs to the algorithms are the sections of mapping parameters (labels, constants, processes and services). The algorithms output the mapping of a section in the memory.

Generation of Executable

Until now, the algorithm outputs are stored in the model which also helps to theoretically evaluate the algorithms. But for a practical evaluation on the hardware, the entire engine control application must be compiled and linked into a target specific executable. Therefore, the next step in the design is to interpret the algorithm outputs and to alter the linking process.

A brief note on compiler and linker is presented here. A *compiler* converts the high level source code (machine independent) to object code (machine specific instructions).

In order to generate instructions which are specific to the target a cross compiler is used. If there are multiple source files, the compiler creates object files for each of source files in the project. A *linker* creates a single executable file from multiple object files by resolving undefined references. To exercise greater control over the linking process, a linker script is used. The main purpose of the linker script is to describe how the sections in the input files should be mapped into the output file, and to control the memory layout of the output file. The sections in the input files plus compiler default sections are controlled by the linker script to generate the efficient memory layout for the target. The outputs of the ILP and greedy algorithms are used to generate header files which contain section names. These header files are included in the linker scripts to perform memory mapping.

3.5 Summary

This chapter has given the details of the techniques used for memory mapping. The terminologies, memory distribution, strategies and the steps to perform memory mapping were introduced. The ILP formulation and the design of greedy algorithms were discussed. Lastly, the practical implementation of ILP and greedy techniques used in the generation of executable were presented. In the following chapter, we apply these techniques to an automotive application. Our experiments involve theoretical evaluations to compare the ILP, greedy techniques and the practical implementation on the ECU hardware.

4

Memory Mapping for an ECU

This chapter describes the experiments conducted to theoretically evaluate the algorithms and to practically measure benefits through implementation on an ECU. This chapter is organized as follows: in Section 4.1, the experimental setup is given which describes the environment, the settings and other necessary details to reproduce the results. Sections 4.2 and 4.3 explain the theoretical and practical evaluation experiments respectively. We summarize the results in Section 4.4

4.1 Environment and Setup

The environment consists of the application, the workstation on which the development and theoretical evaluations are performed, and the target hardware on which the benefits are evaluated. The setup indicates the settings or preconditions to evaluate the algorithms.

4.1.1 Application

The application under consideration is an automotive software and the information related to the number of mapping parameters and their size is shown in Table 4.1.

Mapping Parameter	Number	Size in bytes
Labels	2720	8020
Constants	3023	18116
Processes	1460	257324
Services	1554	135710

Table 4.1: Application details

Distribution Scenario	Method of distribution
Scenario 1	All time synchronous tasks are on Core 0 and all speed synchronous tasks are on Core 1
Scenario 2	Application Software run on Core 0 and Basic Software are on Core 1
Scenario 3	The time synchronous tasks are split between Core 0 and Core 1

Table 4.2: Dual-core task distribution scenarios

The algorithms for memory mapping assume a given task distribution. Depending on the nature of task activation, tasks in automotive systems can be categorized into time synchronous and angle synchronous tasks. The tasks which are periodically activated are termed as time synchronous tasks and the tasks which get activated depending on the engine speed are termed as angle synchronous tasks. As per AUTOSAR [7] and Appendix G (page 66), the software consists of application (Application Software) and its services (Basic Software). Task distribution gives the mapping of a task to a core. There can be many possible task distribution scenarios, but for convenience we consider a dual-core system with three possible task distribution scenarios as show in Table 4.2. The dual-core system is taken as an example because our target hardware available for testing has two cores.

4.1.2 Workstation

The workstation is a PC on which development and theoretical evaluations are conducted. Its configuration along with the tools and solvers used are given in the Table 4.3.

Type	Configuration
Processor	Intel Core 2 Duo
Clock Speed	3 GHz
RAM size	4 GB
Operating System	Windows XP
IDE	Eclipse 3.7.1
Java Version	1.6.0_31
ILP Solver	lp_solve 5.5

Table 4.3: Workstation configuration

4.1.3 Target Hardware

The target hardware which executes the automotive software is a virtual dual-core microcontroller platform based on Infineon architecture. The virtual prototype [31] is chosen as it guarantees the same behaviour as the upcoming hardware and also for its ease of use earlier in the design phase. It consists of two symmetric cores each having their own local RAMs and it has one global RAM in total. Two flash memories with independent read interfaces are provided. Due to high latencies in accessing flash memory, we have instruction and data cache to access code and constant data from the flash.

The hypothetical dual-core system shown in Figure 3.2 (page 17) can be used as a reference for visualizing the hardware. For our algorithms, we assume the memory latency to be known and among the other properties of the hardware, the most relevant inputs to our algorithms are the access times to the RAM and flash. For RAMs, we have a single cycle access to local RAMs and each access to global RAM takes five cycles. In case of flash memories, the accesses are symmetric. In a typical NUMA system, the memories are uniquely addressable and share a single address space. However, in our target hardware this is not possible due to a limitation in RAM accesses. The limitation is that the local RAMs are not shared but only the global RAM is shared. Therefore a message passing mechanism is required to transfer the contents of local RAM to global shared RAM to support remote memory accesses. These restrictions create differences in the ideally assumed theoretical calculations and practically implemented results.

Memory Type	Supported operation	Local latency (cycles)	Remote latency (cycles)
RAM	Read and Write	1	5
Flash	Read	5	10

Table 4.4: Memory access specifications for theoretical evaluation

4.1.4 Settings for Theoretical Evaluation

From the hardware memory specification, we see that the accesses to RAMs are asymmetric and flash memories are symmetric. Since the ILP and greedy algorithms are designed keeping NUMA architecture in mind, we deliberately give asymmetric access times in case of flash by logically dividing them into local and remote flash memories to distribute the placement of parameters. The memory specifications for the theoretical evaluation experiments are as given in the Table 4.4. Local accesses are fast and remote accesses are slow and these differences influence the ILP and greedy heuristics to distribute the flash contents such that all the contents needed by a core are mapped to an exclusive flash memory. This indirectly helps to reduce the contention due to simultaneous flash accesses.

4.2 Theoretical Evaluation

4.2.1 ILP versus Greedy Heuristics

The objectives of the ILP and greedy heuristic are the same - to minimize the total memory access time of an application. However, there are differences in the way they operate. Therefore, we conduct experiments to compare them with respect to:

- Solving time
- Optimality of the solution

We define solving time as the time taken by the algorithm to perform the memory mapping after the inputs are prepared. Solving time is compared by conducting experiments without any constraints on the memory size. Because, under no capacity constraints, the fastest memory is always available and, as the ILP, the greedy heuristic also gives an optimal result. Therefore, with the optimality remaining same, we

can only compare the solving times.

To compare ILP and greedy heuristic for optimality, we conduct experiments by placing constraints on the memory capacity. The ILP always gives an optimal solution, but the result from the greedy heuristic is sub-optimal under capacity constraints due to locally favouring decisions.

Solving Time

The total time complexity of ILP is the complexity of preparing the input and the complexity of solving the ILP. The input preparation of ILP comprises *Cost Calculation* and *Formulation*. The *Cost Calculation* is computed by calculating the number of times a parameter is read or written by each core followed by the computation of the time taken by cores to read memories. *Formulation* involves preparation of objective function and the constraints. When it comes to the solving part, the parameter can be placed in any memory. For an optimal placement, the number decisions made by the ILP solver in the worst case varies exponentially with the number of parameters and memories. Therefore, the total time complexity of ILP can be represented by the following equation:

$$
\begin{aligned}
Complexity_{ILP} &= \mathcal{O}\Big(\overbrace{(\overbrace{Cost\ Calculation + Formulation}^{Input\ Preparation}) + (\underbrace{2^{(Memories \times Parameters)}}_{Solving})}^{}\Big) \\
\mathcal{O}(Cost\ Calculation) &= \mathcal{O}\Big((Parameters \times Cores) + (Cores \times Memories)\Big) \\
\mathcal{O}(Formulation) &= \mathcal{O}\Big(Parameters \times Memories\Big)
\end{aligned}
\tag{4.1}
$$

Likewise, for the greedy heuristic, the total time complexity is the complexity of preparing the input and complexity of solving using the greedy strategy. The input preparation comprises *Affinity Calculation* and *Sorting*. For the *Affinity Calculation* all the parameters are iterated over all the cores to form a list[1] of items, each item contains the parameter and the core which has the most affinity. This list is sorted in descending order of the core affinities using a modified merge sort [28] in the *Sorting* phase. The solving of the greedy heuristic is then performed by placing the parameter

[1]List here represents an array of objects in an object-oriented programming language such as Java

in the memory fastest to the most affine core. Thus, the total time complexity of the greedy heuristic can be represented by the following equation:

$$\begin{aligned} Complexity_{Greedy} &= \mathcal{O}\Big(\overbrace{(Affinity\ Calculation + Sorting)}^{Input\ Preparation} + (\underbrace{Parameters \times Memories}_{Solving})\Big) \\ \mathcal{O}(Affinity\ Calculation) &= \mathcal{O}\Big(Parameters \times Cores\Big) \\ \mathcal{O}(Sorting) &= \mathcal{O}\Big(Parameters \times \log(Parameters)\Big) \end{aligned} \tag{4.2}$$

To observe a noticeable trend in solving times of ILP and greedy heuristics, we do not include the time for input preparation in our measurements. Since the solving time depends on the number of memories and parameters, we vary them one after the other in our experiments to observe the effect.

The effect of varying the number of memories on solving time

For a given application (fixed number of parameters), an experiment is conducted by varying the problem dimensions, without any constraints on the memory capacity. For each dimension of the problem, we have one global RAM and the cores having their local RAMs and flash memories. For example, the hypothetical dual-core scenario of Figure 3.2 (page 17) has one global RAM (Global RAM), two cores (Core 0, Core 1) which have their local RAMs (RAM 0, RAM 1) and flash memories (Flash 0, Flash 1) respectively. Figure 4.1 gives the manner of varying the problem dimensions. The increase in the number of cores also increases the number of memories.

The solving time to place all the mapping parameters in different memories by the ILP and greedy heuristic is reported in the Table 4.5. The variation in solving time is shown in Figure 4.2, from which it is observed that the greedy heuristic is very fast when compared to the ILP. This is because, the greedy heuristic makes a local optimum choice in every iteration. The solving time of both the ILP and greedy heuristic increases with the number of memories. The ILP solving time increases because of the increase in the number of decision variables, while, the greedy heuristic needs more time when it has more memories to search before selecting the fastest memory.

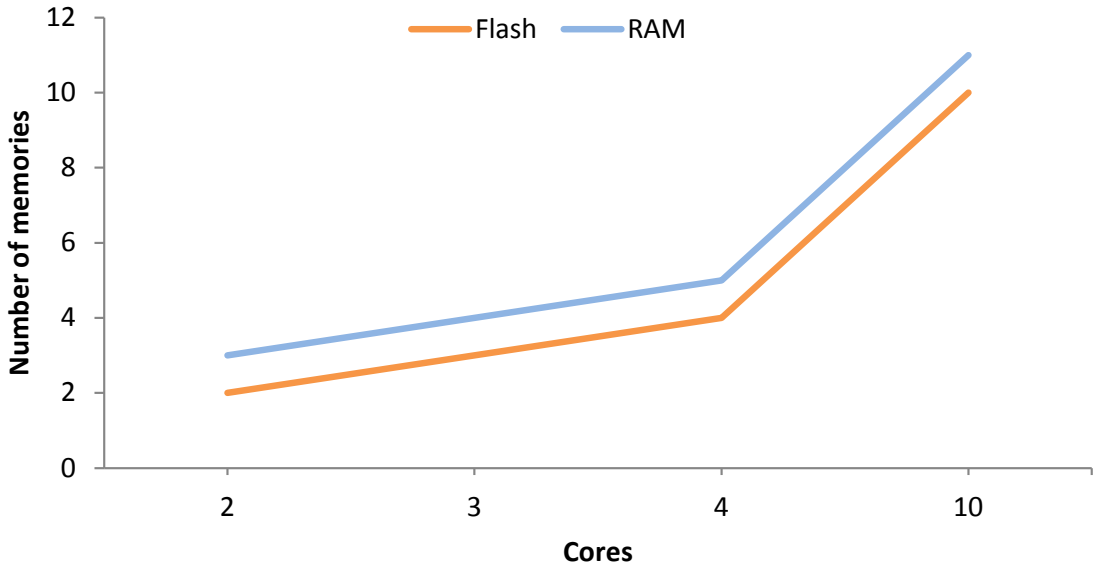

Figure 4.1: Variation of problem size to compare the solving times of the ILP and greedy

Although the solving time complexity (Equation 4.1) of ILP shows an exponential increase with the number of memories, we do not observe this trend because of the placement constraints and relaxed capacity constraints[2]. As per the placement constraints (Equation 3.4) a parameter can be placed in only one memory and according to the capacity constraints (Equation 3.5), the size of parameters cannot exceed the memory capacity. These constraints eliminate a lot of possibilities for the ILP solver, enabling it to show a linear trend rather than exponential. In other words, the presence of placement constraints and relaxed capacity constraints help in achieving a tractable solution by reducing the search space and favouring faster convergence of the branch and bound decisions for the ILP.

However, we expect an exponential behaviour in ILP's solving time for very large number of memories. This is because, the exponential trend is more evident for large values of the exponent but when the exponent is small, the trend is approximately linear or quadratic as seen below. Another instance when there is an exponential increase in solving time is when the memory capacities are too constrained or strict[3]

[2]Relaxed refers to no constraints on the memory capacity
[3]However it is mandatory to have enough memory capacity to fit the parameters

allowing the branch and bound decision tree to grow larger in size. The effect of strict capacity constraint is discussed when we compare the optimality. But before we do that, we study the effect of parameters on the solving time.

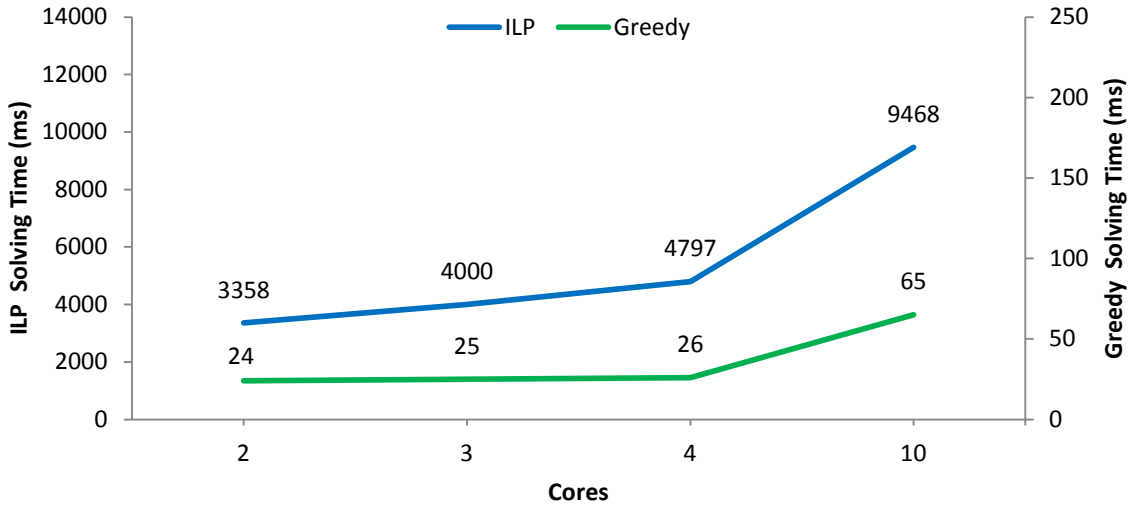

Figure 4.2: Solving time variation of ILP and greedy heuristics for different number of memories

Platform Scenario	Cores	Solving Time (ms)	
		ILP	Greedy
2 Core, 3 RAM, 2 Flash	2	3358	24
3 Core, 4 RAM, 3 Flash	3	4000	25
4 Core, 5 RAM, 4 Flash	4	4797	26
10 Core, 11 RAM, 10 Flash	10	9468	65

Table 4.5: Solving time of ILP and greedy heuristics for different number of memories

The effect of varying the number of parameters on solving time

For a given system (fixed number of memories), we conduct an experiment by increasing the number of labels linearly, without any constraints on the memory capacity. The solving time is then the time taken by the ILP and the greedy heuristic to map all the labels in the memories. The solving time is given in Table 4.6 and its variation

is shown in Figure 4.3 from which it is again observed that the greedy heuristic faster than the ILP.

With the increase in parameters, we observe an increasing trend in the ILP's solving time. This because the parameters appear in the exponent as seen in Equation 4.1. However, the trend is not strictly exponential due to faster convergence of ILP under placement and capacity constraints (relaxed) as mentioned before. The trend seen in the solving time of the greedy heuristic is also not strictly as expected in theory (Equation 4.2). This is because, the algorithms are implemented using the Java programming language and manipulating the data structures using it adds additional complexity to the solving time.

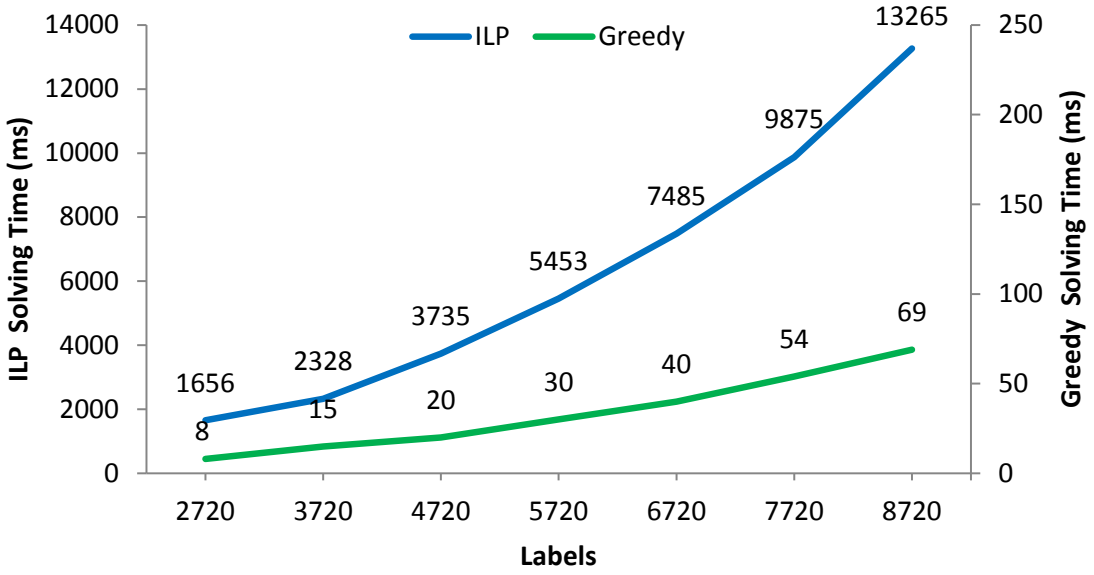

Figure 4.3: Solving time variation of ILP and greedy heuristics for different number of labels

The solving time experiments show that the greedy heuristic very fast compared to the ILP. This favours the greedy heuristic over ILP for problems of large sizes. Now, we conduct experiments to compare the algorithms with respect to the optimality of their solution.

Labels	Solving Time (ms)	
	ILP	Greedy
2720	1656	8
3720	2328	15
4720	3735	20
5720	5453	30
6720	7485	40
7720	9875	54
8720	13265	69

Table 4.6: Solving time of ILP and greedy heuristics for different number of labels

Optimality

The experiments to compare optimality can be conducted only by placing constraints on the memory capacity. This is because, as the ILP, the result of the greedy heuristic is also optimal when there are no constraints on the memory capacity.

The objective of the algorithms is to minimize the total memory access time. A suboptimal solution has a value of the objective function higher than the optimal result. This sub-optimality induces an overhead which is expressed as a percentage, and is given in the Equation 4.3. This is the percentage overhead induced by the greedy heuristic when compared to the ILP.

$$Overhead(\%) = \frac{Objective_{Greedy} - Objective_{ILP}}{Objective_{ILP}} \times 100 \tag{4.3}$$

The greedy strategy works well when there is enough space in the fastest memory. The fastest memory in a NUMA machine is the memory which is local to the core. Considering the hypothetical dual-core scenario as seen in Figure 3.2 (page 17), the local RAMs are RAM 0 and RAM 1. We restrict the local RAMs by constraints and use the global RAM to fit the remaining parameters. However, in case of flash, we do not have a global flash memory. Hence, we approximate the local memory to be the memory of the core in which most frequent tasks run and use the other flash to fit the remaining parameters. For example, if Core 0 runs most of the memory intensive tasks compared to Core 1, then, by placing capacity constraints on Flash 0, we can compare the optimality of greedy heuristics and ILP. The task distribution helps in the identification of the core which runs most of the memory intensive tasks.

We consider local memory capacity to be a small fraction of the total memory requirement for a mapping parameter. The capacity of the local (fastest) memory is reduced in every case and the variation of the overhead is shown in Figure 4.4. The overhead grows with the decrease in the local memory capacity because the sub optimality of the greedy heuristic is more evident when the local memory capacities are less. The experiments are tabulated in Table 4.7. In trial 1 through 6, the local memories are reduced starting from 29% to 14% to observe the increase in overhead. The maximum overhead of 9.24% is encountered when the local memory size is 14% of the total memory size. Further readings were not possible because the problem was too constrained in terms of memory capacity for the ILP and it took impractical amount of solving time. This reason is attributed to the effect of capacity constraints on the branch and bound decisions of the ILP. On one side, the placement constraints help to reduce the total number of combinations checked by the ILP solver to converge to an optimal solution. But, on the other side, if the memory capacity for the fastest memory is too less, the branch and bound decisions take more time because the ILP tries to find a globally best fit for this (constrained) memory. Avissar et al. [8] indicate the practicality of ILP for a small sized application with few memories. However, it is observed that the ILPs are intractable for problems where the capacity constraints are very rigid. Repeating experiments with relaxed constraints has shown faster convergence of the ILP in the experiments considered previously.

Trial	Local Capacity (%)	Objective Value		Overhead (%)
		ILP	Greedy	
1	29%	17806533	18543362	4.14%
2	26%	18061872	18909853	4.70%
3	23%	18404045	19280973	4.76%
4	20%	18566848	19501374	5.03%
5	17%	19395909	20807558	7.28%
6	14%	19737307	21561739	9.24%
7	11%	No solution	22039346	No comparison

Table 4.7: Shows that as local memory capacity is reduced, the overhead induced by greedy heuristic increases. But if the problem has a strict constraint on memory capacity, then the ILP is intractable

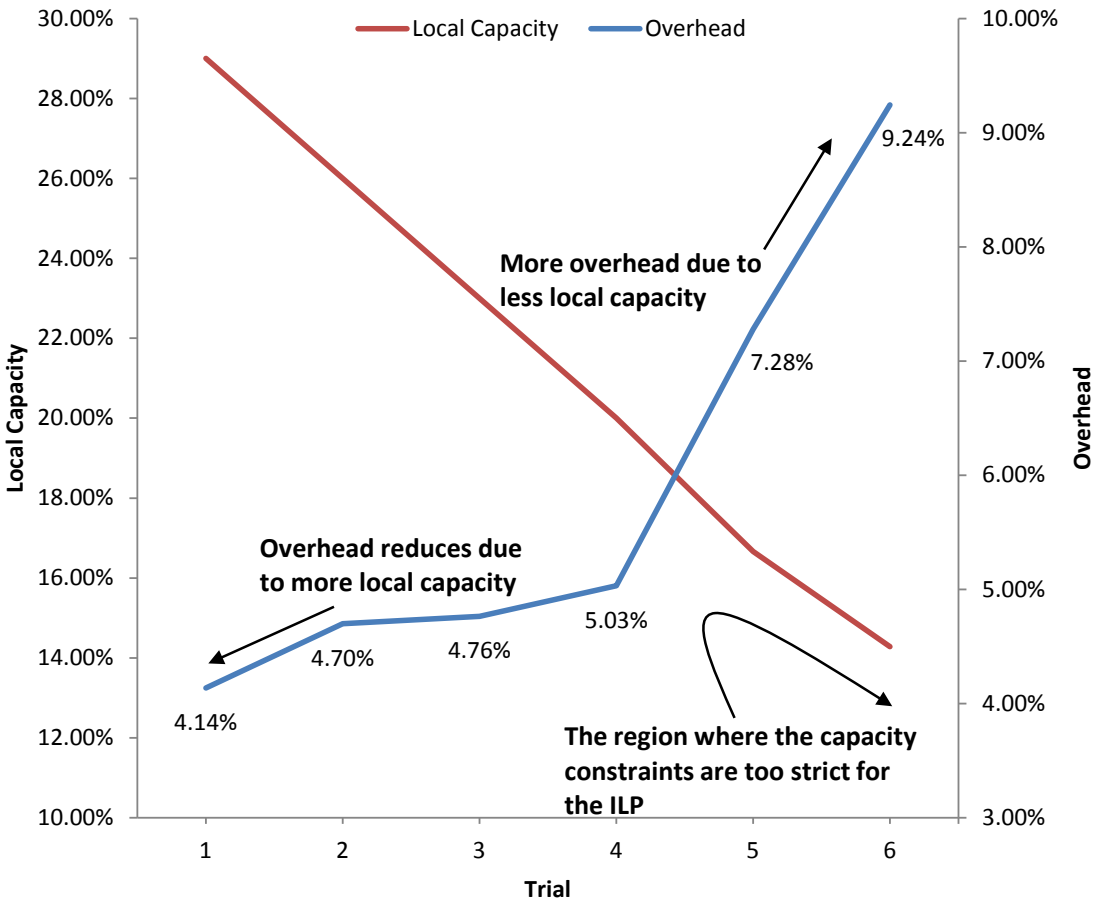

Figure 4.4: The overhead increases with the decrease in the local memory capacity. Having more capacity in the local memory reduces the overhead as seen in the graph. In the experiments, we observe that if the problem has strict capacity constraint, then the ILP is no longer practical.

4.2.2 Selecting an Algorithm

We have presented the ILP and greedy algorithms and here we discuss their selection. As a general approach, for small problems, ILP can be selected first as it provides an optimal solution. Greedy heuristic can be selected whenever ILP becomes intractable due to problem being big or capacity constraints being too strict. When we conducted experiments to compute the optimality, we observed that the greedy heuristic is a practical alternative and even under strict capacity constraints, it induces an overhead

whose magnitude is about 10%, which can be tolerated in practice. It is also worth noting that greedy heuristic is the only alternative to manual placement for a problem to which ILP is intractable.

4.2.3 Algorithm versus Naive Mapping

In this section, we make a theoretical comparison of the algorithms with a naive approach of mapping the labels in RAM. The naive mapping approach is one in which the developer does not want to invest any time in the placement of labels to different memories but chooses only one memory for placement. Given a hypothetical dual-core system as shown in Figure 3.2 (page 17) and an application with *Scenario 1* (from Table 4.2) as task distribution, a naive approach would be to select the mapping of all labels to one of RAM 0, RAM 1 or global RAM. As the baseline, we take the minimum number of cycles which are needed to access the labels which can be computed by assuming a single cycle latency for every memory. The total number of access cycles for baseline, along with the ILP and naive mapping approaches are tabulated in Table 4.8. The memory utilization is the percentage of the time the memory is being utilized which is the defined in Equation 4.4. A memory bandwidth of 80 MHz (80 million cycles per second) is assumed.

$$Memory\ Utilization(\%) = \frac{Total\ access\ cycles}{Memory\ bandwidth} \times 100 \tag{4.4}$$

Case	Total access cycles	Memory Utilization(%)
Baseline	2731879	3.41
ILP	3463099	4.33
All RAM 0	7117063	8.90
All RAM 1	9274211	11.59
All RAM global	13659395	17.07

Table 4.8: Shows how a naive mapping approach requires more access cycles thereby utilizing the memory more than required.

Observation shows that when all the parameters are accessed in a single cycle as given in the baseline, memory is utilized 3.41% of time thus giving the minimum possible memory utilization. ILP reduces the total memory access cycles and results in a memory utilization of 4.33%. This value is greater than the minimum memory

utilization because the shared labels are ultimately placed in one memory, resulting in more than a single cycle access from the remote core. For task distribution *Scenario 1*, Core 0 does more amount of work as it executes the most frequent time periodic tasks. Thus naive approach of mapping everything to RAM 0 results in less utilization when compared to mapping everything in RAM 1. Finally, all the labels mapped to global memory gives the worst mapping leading to a utilization of 17.07%. This experiment shows that naive mapping approaches end up utilizing memory more than required. The ILP approach outperforms the naive mapping approaches by reducing the time for which the memories are utilized.

4.3 Evaluation on a Dual-Core ECU Platform

The implementation is carried out using the concept of *sections* which were introduced in the practical considerations 3.4 (page 30) of the previous chapter. For the practical evaluation, we only use the ILP for distributing the parameters because the problem is small and the memory sizes of the hardware are much bigger than the size of the automotive application (capacity constraints are not strict). The precondition for these experiments is an application which is parallelized for dual-core by distributing the tasks as per *Scenario 1*.

The objective of memory mapping is to reduce the total memory access time. This reduction in access time is reflected in the runtime of the cores. The percentage of time the core spends in effective computation is defined as the core *utilization*. In a dual-core scenario, the overall utilization is the aggregated utilizations of both the cores. We compare the dual-core utilizations before and after memory mapping and represent the improvement as a percentage reduction in utilization as given in Equation 4.5. The utilization when the parameters are distributed is less because the cores spend less time in accessing memory which is due to efficient memory mapping.

$$Improvement(\%) = \frac{Utilization_{undistributed} - Utilization_{distributed}}{Utilization_{undistributed}} \times 100 \qquad (4.5)$$

We express the percentage of *Relative Runtime* as given by Equation 4.6.

$$Relative\ Runtime(\%) = \frac{Utilization_{distributed}}{Utilization_{undistributed}} \times 100 \tag{4.6}$$

Depending on its type, a mapping parameter can be mapped in either flash memory or RAM. We conduct experiments to observe the effect of distributing the parameters among flashes and RAMs.

4.3.1 The Effect of Distributing Parameters in Flash

At first, we conduct experiments to observe the benefit of distributing parameters to different flash memories. The dual-core utilization without any distribution of flash parameters is the reference. Flash content comprises constants, code and we observe the effect of distributing these components by, distributing only constants, distributing only code and distributing both constants and code. To compare the result of algorithm, we also include the result when the code placement is manually performed by distributing it depending on the task placement. Table 4.9 shows the improvements and the relative runtimes for these different mappings. The variation in the relative runtimes are as shown in the Figure 4.5 and we observe that the code distribution using algorithm gives a better reduction in runtime when compared to the manual code placement. It is also observed that the reduction in runtime is more when both code and constants are distributed.

Memory Mapping **(Cache Enabled)**	Improvement (%)	Relative Runtime (%)
No distribution (Single flash)	Reference	Reference 100%
Constant distribution	1.29	98.71
Code distribution (manual)	1.44	98.56
Code distribution	1.83	98.17
Code and constant distribution	2.76	97.24

Table 4.9: In this experiment, the caches are active. The table shows the improvement of distributing parameters in flash memories and the corresponding reduction in runtime when compared to the memory mapping without distribution. In comparison to the manual approach, the improvement is better when the algorithm is used.

From Table 4.9, the improvement is maximal when both code and constants are distributed. The maximum improvement in the runtime achieved from the experiment

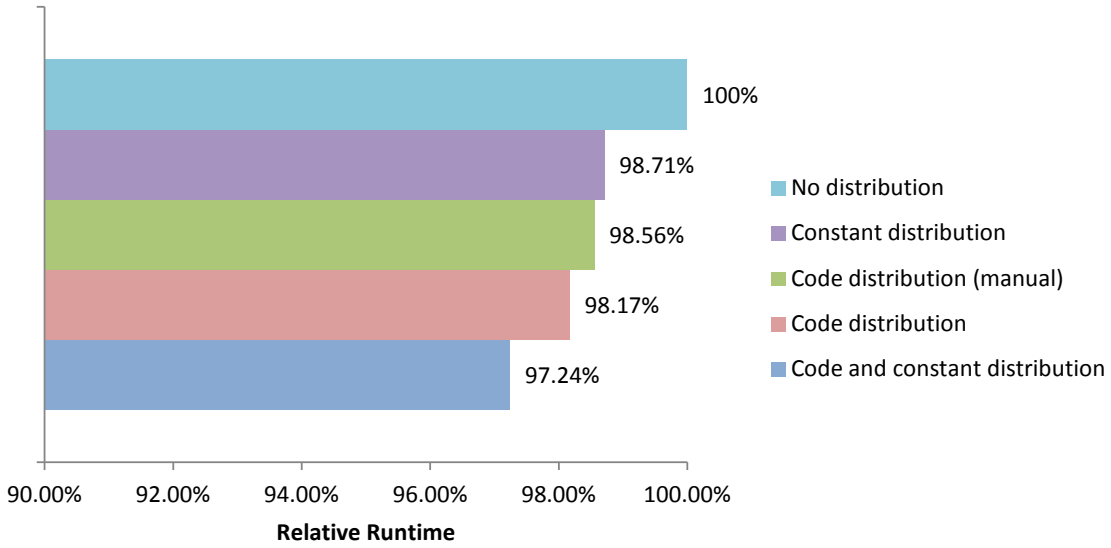

Figure 4.5: The figure shows the reduction in core runtime relative to the runtime when there is no distribution - in the presence of cache.

is 2.76% which is not high enough to justify a real benefit. The reason for this is the presence of cache which is employed to reduce the effective latency of flash memory access. When the cache has a high hit rate, the accesses to the flash memory are reduced and the improvements seen by memory mapping appears to be less because the cache causes the memory access time to be a less significant fraction in the total program execution time. To observe a greater effect, we can either choose a software which uses the flash memory more intensively or we can repeat the experiment by disabling the caches such that accesses to flash are increased. Choosing another application and representing it in our model requires some effort, therefore as an easier alternative, we repeated the experiments by disabling cache to get a higher flash utilization. The measurements are tabulated in Table 4.10 and the variation in the relative runtimes are as shown in Figure 4.6. Firstly, we observe that the distribution of constants has not shown much improvement when data caches are disabled. This is because the constants are accessed in a random fashion and this causes higher flash access penalties when the flash lines are flushed in and out multiple times. However, the improvement is more pronounced in case of code distribution because the code access exhibits high locality of reference such that the flash lines (which act as secondary cache) are effectively utilized. When caches are disabled, the memory access time is a significant

fraction of program execution time. In such situations, memory mapping results in better improvement of about 8.73%.

Memory Mapping **(Cache Disabled)**	Improvement (%)	Relative Runtime (%)
No distribution (Single flash)	Reference	Reference 100%
Constant distribution	0.9	99.10
Code distribution (manual)	6.87	93.13
Code distribution	7.44	92.56
Code and constant distribution	8.73	91.27

Table 4.10: This experiment is conducted by disabling caches. The distribution of parameters in flash now show a greater improvement of 8.73% when compared to the improvement of 2.76% seen when caches were enabled.

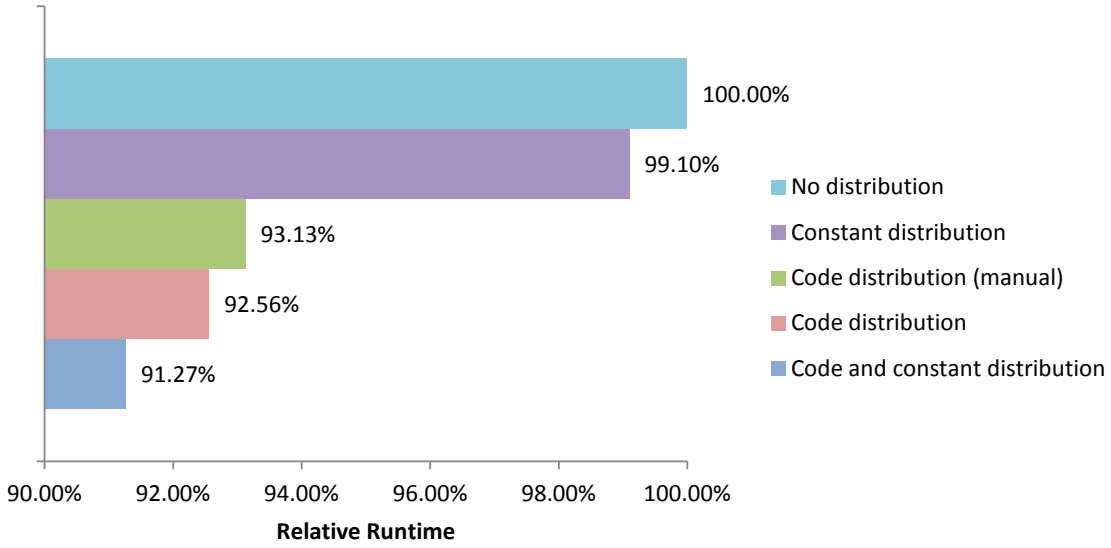

Figure 4.6: The figure shows that when caches are disabled, the benefit of memory mapping is clearly visible as there is more reduction in the relative runtime.

Though our target hardware exhibits an UMA characteristic for flash, we have deliberately used the ILP algorithms with asymmetric access times to get the distribution of flash parameters. The distribution helps in reducing the contention of simultaneously accessing a flash memory. When the accesses to the flash are symmetric, flash contents can be placed to any flash memory but we have shown that, by distributing the flash

contents, we reduce the memory access time by minimizing contention. The benefits of optimal mapping will be better in case of a hardware that exhibits non-uniformity in memory accesses. This is because our placement is performed in a way that the costly remote accesses are reduced.

4.3.2 The Effect of Distributing Parameters in RAM

The stack is the most frequently used portion of RAM and hence its allocation to local memory crucially impacts performance. Since the stack is local to a task, the allocation of stack is performed without using algorithms. Thus, we do not measure the benefit of distributing the program stack but consider this optimization as a base for further distribution.

Distributing parameters in RAM mainly deals with the distribution of labels in different RAMs. As indicated before, we have a limitation in the hardware which does not support remote accesses to local memories. In the case of this hardware limitation, when the tasks are distributed to different cores, the labels needed by both the cores have to be copied from remote memory to local memory via the global RAM. This is the preferred and the best label distribution for the hardware under consideration. The worst distribution will then be the mapping of all labels in the global RAM. The ILP approach implemented as a workaround for the hardware limitation does the following: Firstly, the labels are distributed individually to RAMs and it is worth noting that no labels get placed to global RAM because of its high access cost. Then, the shared labels are mapped to global RAM (as a workaround). Lastly, as per the specifications from AUTOSAR memory mapping [5], new sections are formed by grouping the labels of same size into a section to avoid RAM wastage due to gaps. In this scenario of label distribution using ILP, the number of labels in global RAM (and hence also the utilization) are in between the best and the worst cases mentioned above. If the best and worst case utilizations represent a normalized benefit of 100% and 0% respectively, the benefit of using the ILP (which lies between them) can be calculated by using interpolation as given in Equation 4.7. The Table 4.11 shows these normalized benefits.

$$Benefit_{ILP}(\%) = 100 - \frac{(Utilization_{ILP} - Utilization_{Best}) \times 100}{(Utilization_{Worst} - Utilization_{Best})} \tag{4.7}$$

The memory access to labels in the application are only a fraction of other memory

Case	Normalized Benefit (%)
Worst (All global)	0
ILP (Local and global)	50.48
Best (All local)	100

Table 4.11: This table shows the normalized benefits when all the labels are placed in global RAM, when labels are placed using ILP and when all the labels are in local RAM. It is observed that result of ILP shows a benefit of 50.48% when compared to mapping everything in global RAM.

accesses. Also, if advanced hardware acceleration techniques are used, the impact of accessing labels on the total application runtime will be minimal. This causes the labels accesses to have a little effect on the core utilization. In such a case, comparing absolute core utilization does not indicate a real benefit and therefore we have chosen to normalize the benefit. By interpolating the core utilization of ILP between the best and the worst cases, we observe a benefit of 50.48% which represents the advantage of using ILP when compared to mapping all the labels in global RAM.

4.4 Summary

In this chapter, we have described the environment and the setup in Section 4.1. In Section 4.2 related to the theoretical evaluation, we compared the ILP and greedy heuristics with respect to their solving time and optimality. Our experiments show that greedy heuristic is very fast when compared to ILP which favours the usage of greedy heuristic over ILP for problems of large sizes. It is seen that the ILP always gives an optimal result and the result from the greedy heuristic is suboptimal under capacity constraints which induces an overhead in the magnitude of 10% when compared to the ILP. We observed the effect of capacity constraints on the solving time of ILP and conclude that if the problem has a lot of decision variables, ILP becomes intractable if the capacity constraints are too tight. Hence when it comes to choosing between ILP and greedy, our conclusion is to first use ILP for small problems and choose greedy heuristic when the ILP is intractable. The ILP memory mapping technique was applied to a dual-core ECU in Section 4.3. It is observed that the distribution of parameters in flash results in 2.76% reduction in total runtime when caches were enabled and 8.73% reduction in runtime when caches were disabled. The benefit of

distributing the parameters is more when the accesses to the flash memories are more and further benefits can be expected when flashes exhibit NUMA characteristics as we perform placement to reduce the remote memory accesses that are costly. Due to limitations in the hardware, we had to alter our methodologies to identify the benefits. Finally, we implemented the distribution of labels in RAM using ILP and measured the benefit to be 50.48% when compared to mapping all the labels in global RAM.

5

Refinement using Metaheuristics

The ILP and greedy heuristics described thus far are beneficial. However, their assumptions simplify formulation and disregard the actual hardware behaviour. Further, they have limitations because of contention not being thoroughly addressed. Metaheuristics provide a refinement of earlier techniques and a means to overcome the limitations. A *metaheuristic* is also a heuristic which makes approximations to efficiently solve combinatorial optimization problems. Metaheuristics are classified depending on different criteria [32]. The focus of this chapter is on the design of metaheuristics by means of stochastic optimization. Alternatives to metaheuristics using stochastic optimization can be exact methods like Mixed Integer Non Linear Programming (MINLP) or other metaheuristics based on evolutionary algorithms like GA. As ILP is one of the exact methods which we have already applied and evolutionary concepts are better suited towards exploration problems, we rely on metaheuristics which are stochastic in nature. Starting with a brief introduction to contention, limitations of ILP and greedy techniques, we present two metaheuristics to solve the memory mapping problem.

5.1 An Introduction to Contention

A single port memory can only serve one request per memory access cycle. In such a memory, *contention* is a phenomenon which occurs when there are more than one

requests at the same time. In the case of contention, requests to the memory are delayed and this reflects in increased memory access time for the contended memory. Therefore it can be said that minimizing contention, minimizes the memory access time as well. Previously, the ILP and greedy heuristic worked with the objective of minimizing the total memory access time without directly addressing contention. In this chapter, the metaheuristic works with the objective of minimizing contention.

The notion of contention is explained with reference to the hypothetical dual-core system in Figure 3.2 (page 17). Assuming the case of less latency for local memories and more latency for remote and global memory, the ILP and greedy algorithms fill the local memories until their capacity limit and choose the global memory only if the local memories are full. Further, access times once initialized remain unchanged. But due to existing placements in local memories, there is a possibility of contention which results in increased access time in these memories. Though the assumption of a fixed latency helps in practically getting good results, there is a scope for improvement. A better design is an algorithm which estimates contention in existing placements and diverts future placements to unused (less contentious) memories. The essence of the above idea is captured in two metaheuristics presented in this chapter. Before we discuss them, we highlight the limitations of the previous techniques.

5.2 Limitations of ILP and Greedy Techniques

The limitations in the formulation of ILP and greedy heuristics are due to their simplifying assumptions. The limitations are as below:

- The interconnect conflicts are only indirectly taken care via increased cost to remote memories.
- The assumption of constant memory access time is not always true.
- In case of symmetric accesses to the memories, the ILP and greedy heuristic do not distribute the parameter placement as they disregard contention. Thus the previous techniques work well only for asymmetric accesses.

These limitations are mitigated by using metaheuristics.

5.3 Metaheuristic I

5.3.1 Operating Principle

The fundamental operation of every memory is the same. When a request is sent, the memory takes time to service the request. This service time is constant irrespective of the fact that the request is local or remote. In other words, there is no distinction between the request of local or remote cores as far as memory is concerned. The reason for overall faster or slower access times with local or remote memories respectively are latencies in the bus and possible conflicts in the interconnect. The metaheuristic is designed to overcome the limitations of ILP and Greedy, to capture the real behaviour of memory and interconnects. The operating principle is given below.

After the placement of some mapping parameters, examine the contention. Compute the probability of contention. Depending on the probability of contention, increase the cost of memory access. The metaheuristic now sees the increased cost due to contention and diverts further placements to less contentious memories.

The details of these steps are discussed in the design of metaheuristic.

5.3.2 Design

The steps taken in the design of metaheuristic are as shown in the Figure 5.1. The access costs (in terms of cycles) of all the memories are initialized with costs being symmetric or asymmetric depending on whether the architecture is UMA or NUMA. In the next step, the rate of examining contention is set. This rate corresponds to the number of placements (iterations). The heuristic starts the placement using a greedy approach. Depending on the rate, after a few iterations, the heuristic stochastically computes the contention and alters the access specs. The alteration of access specs is a function of the probability of contention (i.e. higher probability implies higher costs) which depends on the nature of the hardware. The subsequent placements are affected by this increased access cost and the placements are routed to the memory with least cost (incidentally the memory with a low probability of contention). The blocks *Stochastic Computation* and *Alter Access Specs* are described below.

Cores	Total Probability of Contention
2	P_0P_1
3	$P_0P_1 + P_0P_2 + P_1P_2 - 2P_0P_1P_2$
≥ 2 in general	1 - ($P_{no\ core\ access} + P_{exactly\ one\ core\ access}$)

Table 5.1: Stochastic computation of contention

Stochastic Computation

Total contention is stochastically computed. The maximum rate at which a memory can service requests is limited by the memory bandwidth. Thus the probability of a core utilizing a memory is given by the Equation 5.1.

$$P_{Access} = \frac{Num_{Access}}{Max_{Access}} \tag{5.1}$$

Where Num_{Access} is the number of accesses from a core to the memory and Max_{Access} is the maximum possible accesses to memory (limited by memory bandwidth). These probabilities are calculated for all the cores and memories in the system. For example, consider the dual-core scenario of Figure 3.2 (page 17). For RAM 0, let the probability of access from Core 0 and Core 1 be P_0 and P_1 respectively. Clearly the accesses from Core 0 to RAM 0 are independent of the accesses from Core 1 to RAM 0. Since these events are independent, the total probability of contention at RAM 0 is the multiplication of these two probabilities. For every memory, the probability of contention is computed depending on the number of cores as given in Table 5.1.

Alter Access Specs

Depending on the probability of contention at a memory, the access cost to the memory is altered. The amount by which the access specs have to be altered has to be determined by hardware tests. If the probability of contention is high, then the likelihood of contention is more. Therefore highly contentious memories are made costlier for subsequent iterations and the greedy heuristic chooses the memory with least cost which incidentally is also the memory with least contention.

The metaheuristic is given in Algorithm 4. The steps are identical to the greedy heuristic discussed before. The notable difference is the addition of routines for com-

puting contention and altering the access specs. The initialization algorithm required to prepare and sort the list of parameter affinities is given in Algorithm 3.

Algorithm 3 Preparation of the input for greedy metaheuristic

for all *Mapping Parameters* **do**
  for all *Cores* **do**
    CoreParameterAffines ← Add the *Mapping Parameter*, its most affine *Core* along with the affinity to the list
  end for
end for
Sort the list in decreasing order of their affinity

Ensure: *Mapping Parameter* with the maximum affinity is first in the *CoreParameterAffines* list

Algorithm 4 Greedy metaheuristic for *Mapping Parameters*

Require: A list *CoreParameterAffines* as computed in 3
Initialize memory access specs
Initialize Rate of Contention Check
for all *CoreParameterAffines* **do**
  Find the *FastestMemory* available from the *Core*
  if *Parameter* fits into the *FastestMemory* **then**
    Map *Parameter* to → *FastestMemory*
    if Rate of contention satisfied **then**
      Compute Contention
      Alter Access Specs
    end if
  else
    Recursively find the next available *FastestMemory* for the *Mapping Parameter*
  end if
end for

Ensure: That insufficient capacities are reported with an error condition
return In case of no error, return the memory mapping of *Mapping Parameters* to appropriate *Memories*

5.3.3 Assumptions

The following assumptions are made during the design of metaheuristic:

- The values by which we can increase the access specs are determined from hardware tests.

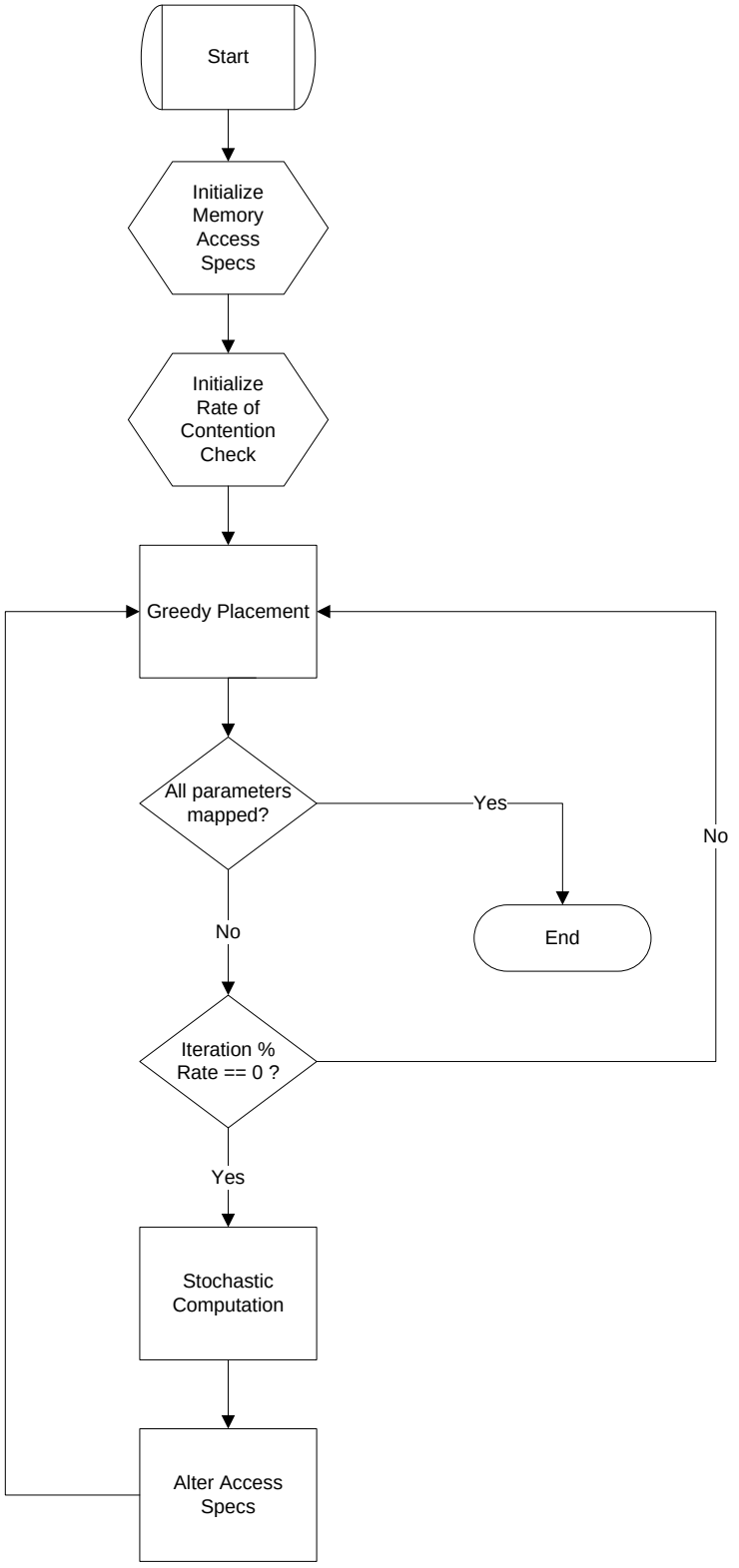

Figure 5.1: Steps in the design of metaheuristic I.

5.4 Metaheuristic II

The metaheuristic disscussed has limitations that the increase in memory access specification has to be determined from tests and the rate at which contention is checked has to be found out using trial and error. Also, all the ILP, greedy and the metaheuristic techniques discussed till now work well only under the assumption that the number of memories are greater than or equal to the number of cores. For example, even in the hypothetical dual-core scenario, we had three RAMs and two flash memory. We face problems when there are more cores but less memories because the access cost from a core to a memory can no longer be the used to distinguish a better placement. This affects the choice of selecting a particular memory for the best placement and therefore we propose another heuristic which performs distribution even when the number of memories are lesser than the number of cores. Such architectures are foreseeable especially when the device vendor stops adding more memories but instead increases the capacity of the existing ones.

5.4.1 Operating Principle

The metaheuristic II works on the following principle:

After initializing the contention probabilities of the memories, a parameter is mapped to the memory with least contention. The contention probabilities for this memory is updated, in subsequent iterations, the memory with least contention is chosen for mapping and the process repeats.

5.4.2 Design

This metaheuristic is shown in Algorithm 5. All the contention probabilities of the memories are initialized to zero in the beginning. In every iteration, a mapping parameter is mapped to the memory with least contention followed by a subsequent re-computation of probabilities. This re-computation increases the contention probabilities after placement. The probability of memory access by a core is as given in Equation 5.1 and the total contention probability for different number of cores can be computed using Table 5.1. Therefore this mechanism creates an equalization in the

memory accesses to different memories and minimizes the contention by distributing the placement to several available memories.

Algorithm 5 Metaheuristic II for Mapping Parameter

```
Initialize contention probabilities of memory
for all Mapping Parameters do
    Map Parameter → Memory with least contention
    Re-compute contention
end for
```

Ensure: That insufficient capacities are reported with an error condition
return In case of no error, return the memory mapping of *Parameters* to *Memories*

5.4.3 Assumptions

This metaheuristic is based on the following assumption:

- Accesses to memories are uniform.

5.5 Evaluation

Due to the difficulties in accurately increasing the memory access time in case of the metaheuristic I, the theoretical evaluation experiments are conducted only using metaheuristic II.

The metaheuristic II aims at reducing the contention by distributing the memory accesses to different memories. To measure the reduction in contention, the metaheuristic is used in the placement of constants and the experiments are repeated by varying the number of flash memories. The application details are the same as given in the Table 4.1 (page 34). It is assumed that the flash memory has a symmetric access of five cycles (note that the number of cycles for this heuristic is not regarded) from all the cores and offers a bandwidth of 10 million accesses per second. In case of symmetric accesses, the distribution performed by the ILP and greedy techniques does not take contention into consideration. To overcome this limitation, the metaheuristic is designed and from evaluation, we show that the contention reduces when this metaheuristic is used.

The aount of flash memories are increased from one to six and this reduces the probability of contention at a memory because the metaheuristic can now distribute the placement among several memories. Further, the probability of contention is equally distributed between the different flash memories. This reduction in contention reduces the memory access time. Table 5.2 shows the contention probabilities at each memory for different number of flash memories.

Flash memory	Contention Probability at each memory (10^{-3})
1	0.96233
2	0.22491
3	0.09982
4	0.05550
5	0.03597
6	0.02422

Table 5.2: This table shows the reduction in contention probability with the increase in the number of flash memories.

5.6 Summary

In this chapter, we have identified the potential drawbacks of the ILP, greedy techniques and proposed two metaheuristics to overcome the limitations. The limitation mainly arises when there is a mixture of symmetric and asymmetric memory accesses in the system. When we have asymmetric accesses, ILP and greedy techniques can be used but this approach does not help in reducing the contention in case of symmetric accesses to memories. The metaheuristic I makes use of stochastic optimization to compute the contention in memories and accordingly increase the cost of memory access for subsequent iterations. This however has a drawback and cannot be applied to scenarios when the number of cores exceed the number of memories. As an alternative to metaheuristic I, we have proposed metaheuristic II which does not rely on memory access cost but stochastically minimizes the total memory contention. In the experiments of placing constants, we evaluated the benefit of metaheuristic II by increasing the number of symmetric flash memories. We see that the metaheuristic II reduces the total memory access time by reducing contention. We have kept the implementation of metaheuristics for our future work.

6

Conclusions and Recommendations

6.1 Conclusions

Memory mapping forms an important step towards optimization in embedded systems. The advances in multi-core hardware have created research opportunities for the development of software techniques to efficiently utilize multiple cores and memories. The software parallelization techniques to utilize different cores are only partial answers to the multi-core challenge because memory now forms the bottleneck for further performance benefits especially when all the cores access the same memory. Further, the non-uniform memory accesses in NUMA machines require an optimal memory layout such that the total memory access time of the application is reduced. Moreover, practical and scalable techniques are needed because the optimal memory mapping problem is NP-complete [23, 21]. In this thesis, we have proposed scalable memory mapping techniques and applied it to a multi-core automotive embedded system to support our hypothesis that the techniques help to reduce the total memory access time.

Our techniques for the memory mapping involve ILP formulation, the design of greedy heuristics and a proposal for using metaheuristics. In our experiments, we have theoretically evaluated the ILP and greedy techniques and it is seen that the greedy technique is about 10% inferior when compared to the ILP. Choosing between ILP and greedy heuristic depends on the number of memories, application parameters and

the capacity constraints of the memory. It is better to choose ILP for small problems and choose the greedy heuristic for problems where ILP is not feasible. To measure the practical benefits, we have implemented the ILP technique on a dual-core ECU platform. The distribution of parameters in flash reduced the total application runtime by 2.76% when caches were enabled and a runtime reduction of 8.73% was observed when the caches were disabled. We expect higher benefits from memory mapping if the flash memories have asymmetric accesses as we perform placement to reduce the costly remote memory accesses. The ILP technique for memory mapping in case of RAM showed a benefit of 50.48% when compared to mapping all the labels in global RAM. We observed that the ILP and greedy heuristics work only when the accesses to the memories are not uniform and we proposed two metaheuristic approaches as add-ons which minimize contention in case of symmetric accesses. However, we could not thoroughly evaluate the metaheuristics for their benefit. Preliminary theoretical evaluation for one of the metaheuristic approach (the Metaheuristic II) shows the reduction of contention with the increase in the number of symmetric flash memories. We have identified the practical implementation of metaheuristics as a part of our future work.

There are benefits observed from the proposed memory mapping techniques. However, applying them for a given use case makes sense only when there is a foreseeable advantage. The introduction of hardware features like caches mask the importance of memory mapping and also its benefit is limited if the labels are less or if they are occasionally accessed. A good indication for the benefit is the fraction of the total execution time spent on memory accesses. If this fraction is a good contribution to the total program execution time, then the benefit from memory mapping techniques will also be higher. The automotive application under consideration was not software intensive and we predict higher benefits in case of softwares which utilize more memory. We have explored the option of increasing the memory utilization by disabling caches for the flash memory, and this has shown a greater benefit. Experiments to prove increased importance of memory mapping by considering a more intensive software can be considered during future work.

6.2 Recommendations

In this section, we highlight the future work and recommend a few approaches which could not be incorporated in this thesis. We see that the metaheuristics are flexible as

they support multiple objectives in their design. The implementation of the proposed metaheuristics is one of our future work which will provide interesting answers to the benefit of using them over the other techniques. These metaheuristics can be easily extended by adding more objectives.

Another possible approach is to use a combination of the above techniques. For example, if we perform an optimal distribution using ILP and then use metaheuristics to reduce contention, the benefits can be higher. Also, to prove improved benefits from the proposed techniques, an intensive software can be chosen and the implemented on the hardware.

We outline a potential problem that arises with respect to multi-core memory mapping here. A mapping parameter shared between two or more cores is ultimately mapped to one memory due to the placement constraint. If the parameter is heavily shared, then the contention at the memory is also higher. This contention was not considered in the metaheuristics but we have potential ideas which can be investigated in our future work. One solution to minimize the contention due to shared data is to duplicate them among several memories. But this is only applicable for read-only data whereas a read-write data needs other mechanisms to ensure consistency. Further, to avoid memory wastage, the duplication has to be controlled and the challenge is to identify the data which benefits the most by duplication. Other techniques to restrict the distribution of tasks to different cores when the tasks share a lot of data can also be employed.

An extension of the proposed memory mapping techniques is a cache aware placement which can be especially performed in the case of flash memory because the access time is higher than RAM. The objective of the cache aware placement technique is to maximize the cache hit rate by proper alignment of data within a flash memory. As a first step, we can use the proposed techniques to identify the placement of mapping parameters to different memories which is then followed by cache aware placement to maximize cache hit.

The task distribution and memory mapping problems are interlinked but there is no fixed order in performing them. One possible approach is to encode the entire task distribution and memory mapping problem in a GA. The GA iteratively improves the candidate solution over generations to identify the best task distribution and its appropriate memory mapping. Though the complexity of the problem is high, efficient encoding of GA can be investigated.

Due to the interdependence between the distribution of task and mapping of memory, we notice that the granularity at which tasks are distributed must be the same as the granularity at which memory is allocated. As an example, if the tasks are distributed depending on functions, then, the memory mapping has to consider the allocation of memory at the level of a function. A finer task distribution followed by a coarser memory distribution and vice versa is not preferred. Therefore, finding the right granularity levels for distributing tasks and memories can be another topic which is worth investigating.

G

Appendix

G.1 Main Working Topics of the AUTOSAR

There are three main topics which are addressed in AUTOSAR:

- Architecture - refers to the Software Architecture including the AUTOSAR Basic Software [1] which serves as an integration platform for hardware independent software applications.
- Methodology - deals with the exchange formats to enable a seamless integration of Basic and Application Software [2] in the ECUs and additionally a methodology stating the use of this framework.
- Application Interfaces - specification of syntactical and semantic interfaces in a view to standardize application software.

Among these topics, the software architecture is more relevant to our work therefore it is described next and additional information can be found on the AUTOSAR homepage [4].

[1] Basic Software - Software responsible for infrastructural functionalities in an ECU [7]
[2] Application Software - Software running on AUTOSAR infrastructure which realizes a defined functionality on an application level

G.2 AUTOSAR Layered Software Architecture

The software architecture is organized in terms of layers. The AUTOSAR layered architecture provides a well defined abstraction of the hardware and a structured interface definition. Figure G.1 shows the various layers of the AUTOSAR software architecture. In Table G.1, the purpose of each of these layers is summarized.

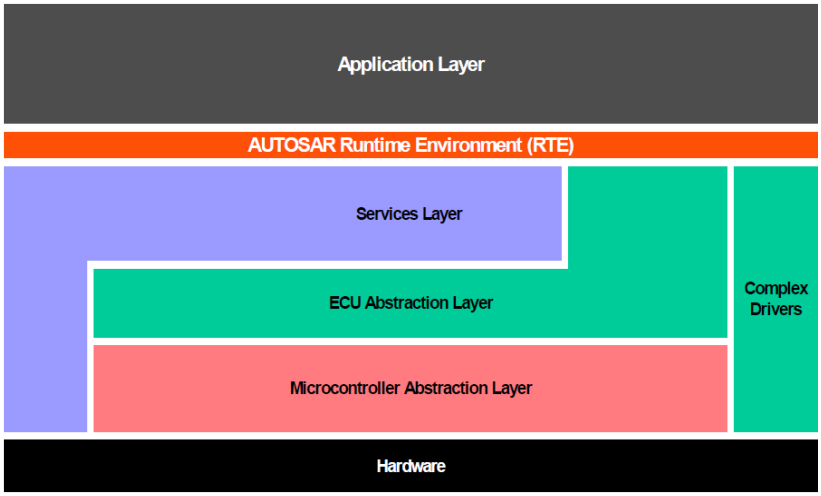

Figure G.1: AUTOSAR: layered software architecture [6]

Layer	Purpose
Application Layer	This is a hardware independent layer which hosts the Application Software Components (SW-C : application running on the AUTOSAR infrastructure)
AUTOSAR Runtime Environment (RTE)	Enables implementation of AUTOSAR SW-Cs independent from mapping to a specific hardware or ECU
Services Layer	Provides basic services to the application layer and also abstracts the microcontroller and ECU hardware from the above layers
ECU Abstraction Layer	Abstracts the ECU layout from the above layers
Complex Drivers	Fulfils the special functional and timing requirements for handling complex sensors and actuators
Microcontroller Abstraction Layer (MCAL)	Makes the upper layers microcontroller independent by abstracting the ECU sensors and actuators

Table G.1: Purpose of AUTOSAR layers

Bibliography

[1] lp_solve, A Mixed Integer Linear Programming (MILP) solver. http://lpsolve.sourceforge.net/, 2008.

[2] *TILE64 Processor: A 64-Core SoC with Mesh Interconnect* (2008).

[3] Antony, J., Janes, P. P., and Rendell, A. P. Exploring thread and memory placement on numa architectures: solaris and linux, ultrasparc/fireplane and opteron/hypertransport. In *Proceedings of the 13th international conference on High Performance Computing* (Berlin, Heidelberg, 2006), HiPC'06, Springer-Verlag, pp. 338–352.

[4] AUTOSAR. Autosar home. www.autosar.org.

[5] AUTOSAR. Autosar specification of memory mapping, 2006.

[6] AUTOSAR. *AUTOSAR Technical Overview R3.0 Rev 1*, February 2008.

[7] AUTOSAR. *AUTOSAR Glossary R4.0 Rev 2*, October 2010.

[8] Avissar, O., Barua, R., and Stewart, D. Heterogeneous memory management for embedded systems. In *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems* (New York, NY, USA, 2001), CASES '01, ACM, pp. 34–43.

[9] Bolosky, W., Fitzgerald, R., and Scott, M. Simple but effective techniques for numa memory management. *SIGOPS Oper. Syst. Rev. 23*, 5 (Nov. 1989), 19–31.

[10] Bolosky, W. J., Scott, M. L., Fitzgerald, R. P., Fowler, R. J., and Cox, A. L. Numa policies and their relation to memory architecture. In *Proceedings of the fourth international conference on Architectural support for program-*

ming languages and operating systems (New York, NY, USA, 1991), ASPLOS-IV, ACM, pp. 212–221.

[11] Broquedis, F., Furmento, N., Goglin, B., Namyst, R., and Wacrenier, P.-A. Dynamic task and data placement over numa architectures: An openmp runtime perspective. In *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism* (Berlin, Heidelberg, 2009), IWOMP '09, Springer-Verlag, pp. 79–92.

[12] Chen, T., Raghavan, R., Dale, J. N., and Iwata, E. *IBM Journal of Research and Development*, 5, 559 –572.

[13] Cox, A., and Fowler, R. The implementation of a coherent memory abstraction on a numa multiprocessor: experiences with platinum. In *Proceedings of the twelfth ACM symposium on Operating systems principles* (New York, NY, USA, 1989), SOSP '89, ACM, pp. 32–44.

[14] D. Cousins, J. Loomis, F. R. P. S., and Tobin, A.-E. The embedded genetic allocator—a system to automatically optimize the use of memory resources in high performance, scalable computing systems. In *IEEE Int. Conf. Systems, Man, and Cybernetics* (1998), vol. 3, p. 2166–2171.

[15] Dandamudi, S. P., and Cheng, S. P. Performance impact of run queue organization and synchronization on large-scale numa multiprocessor systems. *J. Syst. Archit. 43*, 6-7 (Apr. 1997), 491–511.

[16] Freescale. Rationale for multicore architectures in automotive applications. FreeScale Technology Forum, June 2011.

[17] Infineon. Infineon introduces microcontroller multicore architecture for automotive applications. www.infineon.com/cms/en/corporate/press/news/releases/2011/INFATV201110-003.html, 2011.

[18] Infineon Technologies AG. *Memory Access Time in TriCore® 1 TC1M Based Systems*, v 1.1 ed., June 2004.

[19] Intel. Intel core 2 quad. www.intel.com/products/processor/core2quad.

[20] Intel®. Optimizing applications for numa http://software.intel.com/en-us/articles/optimizing-applications-for-numa/, 2011.

[21] Jha, P., and Dutt, N. Library mapping for memories. In *European Design and Test Conference, 1997. ED TC 97. Proceedings* (mar 1997), pp. 288 –292.

[22] Konstadinidis, G. K., Tremblay, M., Chaudhry, S., Rashid, M., Lai, P. F., Otaguro, Y., Orginos, Y., Parampalli, S., Steigerwald, M., Gundala, S., Pyapali, R., Rarick, L. D., Elkin, I., Ge, Y., and Parulkar, I. Architecture and Physical Implementation of a Third Generation 65 nm, 16 Core, 32 Thread Chip-Multithreading SPARC Processor. *Solid-State Circuits, IEEE Journal of 44*, 1 (2009), 7–17.

[23] Kumar, T. R. *On-Chip Memory Architecture Exploration of Embedded System on Chip*. PhD thesis, Indian Institute of Science, 2008.

[24] Larowe, Jr., R. P., and Schlatter Ellis, C. Experimental comparison of memory management policies for numa multiprocessors. *ACM Trans. Comput. Syst. 9*, 4 (Nov. 1991), 319–363.

[25] Larus, J. Spending moore's dividend. *Commun. ACM 52*, 5 (May 2009), 62–69.

[26] MathWorks. Simulink http://www.mathworks.com/products/simulink/index.html.

[27] Navarro, A., Zapata, E., and Padua, D. Compiler techniques for the distribution of data and computation. *IEEE Trans. Parallel Distrib. Syst. 14*, 6 (June 2003), 545–562.

[28] Oracle. Java^{TM} platform, standard edition 6 api specification http://docs.oracle.com/javase/6/docs/api/. Refer sort method in Collections class.

[29] Panda, P. R., Dutt, N. D., and Nicolau, A. Architectural exploration and optimization of local memory in embedded systems. In *Proceedings of the 10th international symposium on System synthesis* (Washington, DC, USA, 1997), ISSS '97, IEEE Computer Society, pp. 90–97.

[30] Panda, P. R., Dutt, N. D., and Nicolau, A. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. *ACM Trans. Des. Autom. Electron. Syst. 5*, 3 (July 2000), 682–704.

[31] Synopsys. Virtual prototyping. Web Article. http://www.synopsys.com/systems/virtualprototyping/Pages/default.aspx.

[32] Talbi, E.-G. *Metaheuristics: From Design to Implementation.* John Wiley & Sons, Inc., 2009.