Compile Time Task and Resource Allocation of Concurrent Applications to Multiprocessor Systems

Nadathur Rajagopalan Satish

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-19
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-19.html

January 29, 2009

Compile Time Task and Resource Allocation of Concurrent Applications to Multiprocessor Platforms

by

Nadathur Rajagopalan Satish

B.Tech. (IIT Kharagpur) 2003

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Engineering - Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Kurt Keutzer, Chair
Professor John Wawrzynek
Professor Alper Atamtürk

Spring 2009

The dissertation of Nadathur Rajagopalan Satish is approved:

_____

Chair Date

_____

Date

_____

Date

University of California, Berkeley

Spring 2009

Compile Time Task and Resource Allocation of Concurrent Applications to Multiprocessor Platforms

Nadathur Rajagopalan Satish

Abstract

Compile Time Task and Resource Allocation of Concurrent Applications to Multiprocessor Platforms

by

Nadathur Rajagopalan Satish

Doctor of Philosophy in Engineering - Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Kurt Keutzer, Chair

Single-chip multiprocessors are now commonly present in both embedded and desktop systems. A key challenge before a programmer of modern systems is to productively program the multiprocessor devices present in these systems and utilize the parallelism available in them. This motivates the development of automated tools for parallel application development. An important step in such an automated flow is allocating and scheduling the concurrent tasks to the processing and communication resources in the architecture. When the application workload and execution profiles are known or can be estimated at compile time, then we can perform the allocation and scheduling statically at compile time. Many applications in signal processing and networking can be scheduled at compile time. Compile time scheduling incurs minimal overhead while running the application. It is also relevant to rapid design-space exploration of micro-architectures.

Scheduling problems that arise in realistic application deployment and design space exploration frameworks can encompass a variety of objectives and constraints. In order for scheduling techniques to be useful for realistic exploration frameworks, they must therefore be sufficiently extensible to be applied to a range of problems. At the same time, they must be capable of producing high quality solutions to different scheduling problems. Further, such techniques must be computationally efficient, especially when they are used to evaluate many micro-architectures in a design space exploration framework.

The focus of this dissertation is to provide guidance in choosing scheduling methods that best trade-off the flexibility with solution time and quality of the resulting schedule. We investigate and evaluate representatives of three broad classes of scheduling methods: heuristics, evolutionary

algorithms and constraint programming. In order to evaluate these techniques, we consider three practical task-level scheduling problems: task allocation and scheduling onto multiprocessors, resource allocation and scheduling data transfers between CPU and GPU memories, and scheduling applications with variable task execution times and dependencies onto multiprocessors. We use applications from the networking, media and machine learning domains to benchmark our techniques. The above three scheduling problems, while all arising from practical mapping concerns, require different models, have differing constraints and optimize for different objectives. The diversity of these problems gives us a base for studying the extensibility of scheduling methods. It also helps provide a more holistic view of the merits of different scheduling approaches in terms of their efficiency and quality of solutions produced on general scheduling problems.

Professor Kurt Keutzer
Dissertation Committee Chair

Dedicated to my family.

Contents

List of Figures

List of Tables

Acknowledgments

I thank my advisor, Professor Kurt Keutzer for guiding me through my graduate life at Berkeley. He has inspired me in various ways. As a researcher, I am continually amazed at his clarity of thought and his ability to get the fundamental idea behind a problem with just a few questions. The emphasis he places on the practical aspects of solving a problem will continue to inspire and guide me in my future career. His focus on communication skills has left me a much better public speaker than when I came to Berkeley. As an advisor, he has always struck the right balance between allowing students to perform independent research versus guiding them to ensure they pick topics of practical interest. It has been an honor to work with him and I look forward to continuing to collaborate with him in the years ahead.

I thank Professor John Wawrzynek and Professor Alper Atamtürk for serving on my dissertation and qualifying examination committees. They have provided me with valuable guidance and feedback regarding this dissertation. Professors Wawrzynek's course on reconfigurable computing was an inspiration for the first portion of this work. Professor Atamtürk taught me two courses on computational and linear optimization, and his courses provided the spark and a basic foundation for a significant part of this research.

I thank Professor Ras Bodik for having served on my qualifying exam committee. He provided valuable feedback to me on my research topic that has helped shape the direction of my work.

I thank all my professors who have taught me various courses here at Berkeley. They have not only imparted me knowledge of the subject matter, but have also enabled me to think objectively and critically about research in various fields.

I thank my direct collaborators in my research work: Yujia Jin, William Plishker, Kaushik Ravindran and Narayanan Sundaram. I would especially like to thank Kaushik Ravindran, who has closely collaborated with me for most of my research career at Berkeley. We studied different models and optimization methods to solve mapping problems.

I thank the present and past members of the MESCAL research group, which has been renamed the PALLAS group. These include: Jike Chong, Bryan Catanzaro, Matt Moskewicz, Andrew Mihal, Scott Weber, Niraj Shah, Dave Chinnery, Christian Sauer, Matthias Gries, Chidamber Kulkarni and Martin Trautmann. They are a really smart group of people. They have sparked many an idea.

I thank other students in the DOP Center for having made life more fun. These include (but are not limited to): Abhijit Davare, Animesh Kumar, Arindam Chakrabarti, Arkadeb Ghosal, Donald Chai, Douglas Densmore, Kelvin Lwin, Krishnendu Chatterjee, Mark McKelvin, Nathan Kitchen,

Qi Zhu, Satrajit Chatterjee and Trevor Meyerowitz. I have become good friends with most of this group. I remember a few sightseeing trips we went on together - those were good fun!

I would like to thank the members of the EECS administrative staff: Mary Byrnes, Ruth Gjerde, Jontae Gray, Patrick Hernan, Cindy Keenon, Brad Krepes, Ellen Lenzi, Loretta Lutcher, Dan MacLeod, Marvin Motley, Jennifer Stone, and Carol Zalon. They have always helped students navigate around the various procedures and bureaucracy involved in graduate life. My thoughts and prayers remain with Ruth.

A shout-out to my former and current apartment-mates: Shariq Rizvi, Rahul Tandra, Pankaj Kalra, Susmit Jha and John Blitzer: the lunches and dinners, the conversation, watching sports and movies - all have now become a regular part of my life. I know I'll miss all of the camaraderie.

To my parents and sister - your encouragement, effort and sacrifices are what made this possible. I cannot possibly thank you enough.

Chapter 1

Introduction

There has been a major shift in the computing landscape due to the advent of devices with multiple processors on a single chip. The shift towards single-chip multiprocessors is motivated by the simple fact that while Moore's law has continued to allow the doubling of transistors on a die every 18-24 months, these transistors can no longer be effectively used to increase performance of a single processor. The prohibitive increase in power consumption of single processor designs and the complexity of designing such processors are the key reasons that prevent single core performance from scaling [Olukotun and Hammond, 2005] [Borkar, 1999]. The alternative method to utilize the additional transistors that are made available in each process generation is to instantiate multiple processors on the same chip. In keeping with Moore's law, it is expected that the number of processors per die with double with each process generation [Asanovic *et al.*, 2006].

As of 2008, single chip multiprocessors have become firmly established in the server, desktop and embedded markets. Some of the early trends towards chip multiprocessors occurred in the embedded market. These early chip multiprocessors were driven by the need to satisfy the ever-increasing computational requirements of embedded applications while not sacrificing the programmability of the platform [Horowitz *et al.*, 2003] [Masselos *et al.*, 2003]. As a consequence, programmable multiprocessors with high computational capability were designed in an attempt to exploit the concurrency naturally present in many embedded applications. Such multiprocessors have remained useful in light of the limits to which single processor performance can be improved.Some examples of programmable multiprocessors in the embedded market include the IXP network processors from Intel [Intel Corp., 2002], the CRS-I Metro Chip from Cisco [Eatherton, 2005], the Cell processor by the IBM-Sony-Toshiba consortium [IBM Corp., 2007] and Digital Signal Processors (DSPs) from TI, Analog Devices and Picochip. It is to be noted that this is by no

means an exhaustive list and other companies have also brought out or are in the process of bringing out programmable multiprocessor chips.

In the general-purpose market, Intel and AMD started marketing dual-core processors for the desktop market around 2004-2005. The trend in the general purpose market has been from multi-core to many-core: from dual-, quad- and eight-core chips to chips with 32 cores, 64 cores and beyond. This is a natural consequence of Moore's law, and the number of processors on a die can be expected to continue to scale to hundreds of cores. The advent of Graphics Processing Units (GPUs) from NVIDIA and ATI/AMD and the forthcoming Larrabee processor from Intel are early indicators of this trend.

At the same time, multiprocessor designs have also been implemented on Field Programmable Gate Array (FPGA) platforms. FPGAs allow for the instantiation of a network of processing elements, logic blocks and memories using Lookup Tables (LUTs) on the fabric. One of the key advantages of FPGA based systems is its flexibility to customize the architecture to suit different application requirements. The primary FPGA vendors, Xilinx and Altera, offer tools to help design multiprocessor networks on their platforms [Xilinx Inc., 2004] [Altera Inc., 2003]. FPGA based multiprocessor systems have primarily been used as a emulation platform for evaluating multiprocessor architectures, and as a research tool for evaluating the scalability of multiprocessor systems [Wawrzynek *et al.*, 2007].

Programmable single-chip multiprocessors are now an established trend in both the embedded and general-purpose markets. Such processors offer the potential to allow application performance to keep scaling with process generation. However, in order to harness this potential, programmers must be able to effectively utilize the resources available on these devices. This is often a challenging task on multiprocessors which can have a complex set of architectural features to exploit.

1.1 Challenges in the use of single chip multiprocessors

The key challenge before a programmer is to utilize the parallelism inherent in modern platforms. It is a challenge to keep the increasing numbers of processors on modern chips busy. One approach to utilize the parallelism in the hardware is to run completely independent applications in parallel on different processor cores. However, the number of applications that run simultaneously at a point of time is currently limited in most typical embedded or desktop applications. This is not a scalable solution as the number of processors increases. This means that each single application must be capable of utilizing multiple cores. Fortunately, many applications, especially in the net-

working, multimedia and gaming fields as well as emerging desktop applications such as the Intel RMS application suite [Chen *et al.*, 2008], are inherently concurrent. The concurrency present in such applications may be at a coarse level with multiple threads of control that coordinate among themselves, or at a finer level within a single thread of execution. The first level of concurrency is called task-level or thread-level concurrency. A program that uses threading libraries like Pthreads uses task-level concurrency. Each task may also exhibit data-level concurrency if the task can process many units of data in parallel. The hardware may also support data-level parallelism when each processor is capable of processing more than piece of data in parallel, typically using vector/SIMD units.

The programmer must then map the task and data level concurrency in the application to the parallelism in the architecture. The aim of the programmer is to produce high-performance implementations on the target platform. At the same time, the programmer must be able to develop the implementation productively. These dual goals are often in conflict. The ability of the programmer to come up with high-performance parallel implementations is hindered by many factors. First, there is usually no formal technique for application programmers to express the task and data level concurrency in the application. There is similarly no facility to express the parallelism in the architecture. Second, the granularity and extent of concurrency in the architecture may not match the architectural features of the platform. For instance, different tasks in the application will usually not take the same amount of time to execute on the target platform. In such a case, it is the responsibility of the programmer to manually balance the computations performed on different processors to ensure that no single processor becomes the computational bottleneck. In order to ensure that communication does not become a bottleneck, tasks that communicate a lot of data with each other must be allocated to processors are physically close on the chip. Third, the architecture may impose a varied set of restrictions on the possible mapping of the application. For instance, there may be restrictions on which processors can communicate with each other. The size and connectivity of local memory could also restrict the distribution of tasks among the processors. Finally, the presence of heterogeneity in terms of the processing elements or memory access times further complicates the mapping problem.

In view of these challenges to producing high-performance implementations, programmers often resort to coding applications at a fairly low level in order to fine-tune their implementations. Such a programming practice is highly unproductive and error-prone. Moreover, as platforms change, the program must often be recoded to take advantage of the new architectural features. These reasons motivate an automated approach to programming multiprocessor chips.

From the above discussion, we see that there is a big gap between the concurrency present in the application and the low-level architectural features in the platform. This gap is called the implementation gap [Gries and Keutzer, 2005]. In order to utilize the hardware efficiently and productively, it is critical to automate the process of bridging this implementation gap.

1.2 Bridging the implementation gap

The implementation gap between a concurrent application and the architectural platform arises due to the obstacles in modeling the application and architecture and mapping the application onto the architecture. There are three main aspects to successfully bridge this gap:

- Construct an application description that exposes the concurrency in the application
- Create an architectural model that captures the performance related hardware features and resource limitations
- Map the application description to the architecture model to optimize for a given metric

The popular Y-chart method for Design Space Exploration (DSE) advocates a separation between these three aspects of bridging the implementation gap. The basic flow of the Y-chart approach is shown in Figure 1.1. The approach starts with independent descriptions of the application model and the architecture model. The *mapping step* then is responsible for binding the application onto the architecture. The results of the mapping are then evaluated and iterative refinements are then made either to the way the application is represented (or a different algorithm is chosen, the choice of architecture or the mapping step. This continues until the desired performance goal (and/or cost) is achieved.

The application and architecture models used in the Y-chart approach will differ depending on the granularity of concurrency that the programmer wishes to exploit. In this dissertation, we limit our attention to the mapping issues revolving around the task-level parallelism in the application. In such a case, a popular representation for an application is a *task graph* that exposes the concurrent tasks, the input and output data of these tasks and the dependencies between the tasks. The architecture model is typically a network of processors that exposes the communications between the processors and the memories that connect to them. The *mapping step* is then responsible for (a) allocating the tasks to the processors, (b) allocation of data to the memories, (c) allocation of the communication and data transfers between tasks to the communication links in the architecture,

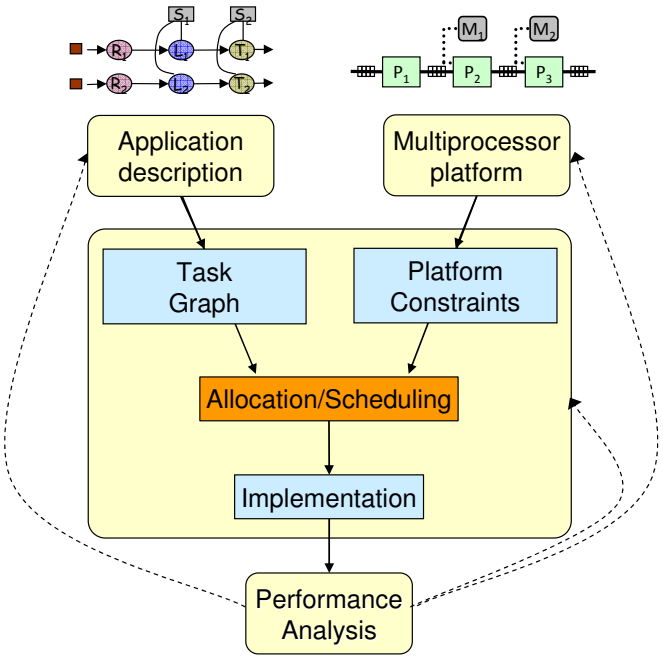


Figure 1.1: The Y-Chart approach for deploying concurrent applications.

(d) scheduling the tasks in time and (e) scheduling the communication and data transfers in time. While performing this mapping, restrictions on the architectural topology, task clustering, preferred allocation of tasks to processors, limited amount of memory, constraints on communication bandwidth, heterogeneity in processors/communication links etc. must be respected. The objective of the mapping problem is to optimize for performance, power, cost or a combination of such metrics. The mapping problem in general will necessarily be a complex and potentially multi-objective optimization problem.

For applications that have a combination of task and data level parallelism, the models described above express only the task-level parallelism directly. However, this does not mean that data-level parallelism cannot be leveraged: each task can have data-parallel operations inside its definition and can be mapped onto the SIMD/vector units inside of a single processor. In this case, after the allocations of tasks to processors is complete, an additional compilation step must be performed to extract the data-level parallelism. Details of such approaches can be found in [Gries and Keutzer, 2005, Chapters 3-4].

1.3 The mapping step

The mapping step is a key step in enabling efficient design space exploration. The mapping step involves solving a combinatorial optimization problem over a large design space under many resource constraints. It is difficult to manually find good solutions to the mapping problem. Manual efforts to solve the problem revolve around finding solutions that appear intuitive to the programmer. However, with the growing complexity of applications and architectures, it becomes increasingly likely that such solutions will prove sub-optimal [Gries, 2004]. The need of the hour is hence to find automated techniques to solve the mapping problem.

A number of approaches have been described to map the task level concurrency in the application to a multiprocessor network. Some of the early work was carried out in the Operations Research community in the context of mapping jobs to be done in workshops onto multiple machines. The techniques developed there came to be known as *scheduling methods* [Hall and Hochbaum, 1997]. There have been many variants of the scheduling problem that have been considered. Graham et al. [Graham *et al.*, 1979] introduced a 3-field $\alpha|\beta|\gamma$ classification of scheduling methods onto where α denoted the machine (target platform) characteristics, β the job characteristics and γ the scheduling characteristics. The different possibilities for the target machine α cover the number of machines and whether the machines were identical (homogeneous). The β parameter covers many different job characteristics, including whether jobs can be preempted, whether jobs consume other resources other than machine time, whether jobs are independent or have dependencies, and whether tasks have differing execution durations (often called *execution times*) on the same machine. The γ parameter indicates the optimality criterion of the scheduling problem, with the possibilities being the end-to-end completion time (or *makespan*, represented as C_{max}), sum of task lateness with respect to task deadlines, or the number of tasks that do not meet their deadlines. Similar classifications have also been proposed for multiprocessor scheduling problems. The surveys by [Kwok and Ahmad, 1999b] and [Casavant and Kuhl, 1988] introduce a taxonomy of scheduling problems for multiprocessor scheduling.

In this dissertation, we shall concern ourselves with scheduling problems related to mapping concurrent tasks onto multiprocessors. A number of such problems fall under the $R|prec|C_{max}$ classification according to Graham. The classification can be explained as follows: parameter R indicates that the target architecture has multiple heterogeneous processors, parameter $prec$ indicates that the tasks have precedence constraints and parameter C_{max} indicates that the optimization criterion is the makespan of the system. A number of additional resource constraints are also usually

required to solve practical problem instances. In addition, we also consider a different scheduling problem that minimizes the total time required for data transfer between tasks. Different variants of multiprocessor scheduling problems with differing constraints and optimization metrics have been studied in the literature. In the following sections, we shall list the theoretical complexity results and practical algorithms developed for multiprocessor scheduling.

1.3.1 Complexity of the scheduling problem

Many variants of the scheduling problem have been proven to be NP-COMPLETE in the scheduling literature. For the special case of $1|prec|C_{max}$ (single machine), there exist polynomial-time solutions. However, the problem becomes NP-hard when there is more than one processor, even if there are no precedence constraints between tasks [Lenstra *et al.*, 1977]. The general problem of finding a task schedule that minimizes the makespan of the system ($R|prec|C_{max}$) is a generalization of this problem and is therefore also NP-hard. In fact, a number of these problems have also been proven to be strongly NP-hard in the sense that no polynomial-time approximation algorithms are possible unless P=NP [Garey and Johnson, 1978]. The decision versions of these problems have also been shown to be in the class NP, and are hence NP-complete [Brucker, 2001].

The result that scheduling problems are NP-complete means that we cannot expect to obtain optimal or near-optimal solutions for all possible instances of the scheduling problem. However, this does not automatically mean that there do not exist viable practical approaches to solve these problems. This is because NP-completeness is only an indication of the worst-case complexity of solving a problem, and not of the difficulty of solving individual practical instances of the problem. For the multiprocessor scheduling problem, there have been a variety of computationally efficient techniques that can solve practical instances of various scheduling problems. The surveys of [Kwok and Ahmad, 1999b] and [Casavant and Kuhl, 1988] provide a taxonomy of approaches that have been used to solve the multiprocessor scheduling problem. At the top level, such classifications distinguish between compile-time (or static) scheduling methods and run-time (or dynamic) scheduling methods. Compile-time methods make scheduling decisions before the actual execution of the application. These decisions are based on reliable models of the application in terms of the tasks, dependencies between tasks and performance characteristics of individual tasks. The resulting schedule is then used to implement the application. Once the scheduling decisions have been made, they are not changed during the execution of the application; hence compile-time methods are also often called *static* methods. In contrast, run-time methods make scheduling decisions as the

application executes. They do not assume knowledge about the application characteristics such as the task graph structure or execution times of tasks before execution. As the application executes, these algorithms maintain a list of all tasks that are ready to execute at a particular point of time and then schedule these tasks on-the-fly to processors that are not busy at that time.

1.3.2 The case for compile-time scheduling

In this dissertation, we concentrate on the use of compile-time techniques for scheduling. Compile-time techniques have a key advantage in that all scheduling activities are done before the application executes and hence there is almost no scheduling overhead (except the cost of implementing the schedule) while the application runs. On the other hand, run-time dynamic scheduling algorithms must devote some of the computational capability of the platform to do the scheduling. When the application workload and execution characteristics are known at compile time, compile-time scheduling has clear benefits over run-time scheduling.

A second disadvantage of run-time schedules is that they do not take into account the global nature of the scheduling problem. Since dynamic scheduling techniques do not assume knowledge about tasks that will be available to run in the future, they cannot take such tasks into account while making scheduling decisions. They are thus inherently "short-sighted" and offer no guarantee of global optimality. On the other hand, compile-time techniques can use knowledge about future tasks and their execution characteristics to produce more optimal solutions to the scheduling problem.

In the context of design space exploration for multiprocessor systems shown in Figure 1.1, dynamic evaluation of designs would necessitate the elaboration of an entire micro-architecture, synthesis of that micro-architecture and re-targeting the application to that micro-architecture. In many cases, building an executable model may be very expensive during exploration. Further, re-targeting the application involves either the development of a compiler for the new platform or a manual reimplementation of the program. These options are usually expensive in terms of design effort and time. In such cases, compile-time methods that use models of the application and architecture rather than a physical instantiation of the system become useful. Compile-time methods can quickly prune a large part of the design space without the need for expensive synthesis [Gries, 2004].

Compile-time techniques rely on the knowledge of application workload and execution characteristics of tasks at compile-time. Several applications in the signal processing and network processing domain are amenable to compile-time scheduling [Sih and Lee, 1993] [Shah *et al.*, 2004].

Models of the execution characteristics of the application are derived from analytical evaluations of application behavior, or through extensive simulations and profiling of application run time characteristics on the target platform.

However, in certain applications, the tasks that need to be performed may depend on application inputs which are only known at run-time. For such applications, it is not possible to model the application at compile-time. In such cases, dynamic scheduling approaches must be used.

1.3.3 Methods for Compile-time Scheduling

Compile-time scheduling algorithms fall into three major categories: heuristics, randomized algorithms and exact methods. These algorithms have different tradeoffs in terms of the three metrics: the quality of solution they produce, computational efficiency of the scheduling method and the extensibility of the method. We describe these metrics below.

Metrics to choose scheduling methods

- Quality of scheduling results: defined by how close the scheduling method can approach the optimal solution to the scheduling problem.
- Computational Efficiency: defined by how the computational and memory requirements of a scheduling method.
- Extensibility: defined by how easily a scheduling method can be modified to satisfy a diverse set of constraints.

While it is evident that a good scheduling method must produce high-quality results with reasonable computational efficiency, some explanation is due for the third metric. The extensibility of a scheduling algorithm is a way of denoting how easily practical constraints on the mapping problem can be accommodated in the method. For instance, the architecture may impose constraints related to topology restrictions, memory size limits and communication bandwidth limits that restrict the space of valid mappings. Restrictions may also arise die to application constraints on task affinities to processors and maximum communication overheads. Further, the programmer may manually add in constraints to offer insight into the scheduling problem over the course of design space exploration. An extensible scheduling method will ensure that such constraints can be easily added while solving the problem.

The three metrics mentioned above for comparing scheduling algorithms are often at odds with each other. Therefore, algorithms generally prioritize one or two of the three metrics at the expense of the rest. Techniques for compile-time scheduling range from heuristics that emphasize the computational efficiency of the scheduling method to exact techniques that emphasize the quality of the solution obtained. We describe below the individual trade-offs offered by different compile-time scheduling approaches.

Heuristic Approaches

Heuristic techniques are computationally efficient algorithms that attempt to find acceptable solutions to scheduling problems. The exact nature of heuristics used depends on the objective and constraints of the scheduling problem. Surveys of heuristic approaches for the $R|prec|C_{max}$ scheduling problem outlined earlier have been reported in [Kwok and Ahmad, 1999b]. The most commonly used heuristics are list-scheduling based algorithms [Sih and Lee, 1993] [Hu, 1961] [Coffman, 1976]. The value of a heuristic is that is uses some pre-existing knowledge of a problem to find a good solution while exploring a very small portion of the search space. As a consequence, heuristics are in general a good choice only when they have been developed for the particular scheduling problem at hand. For instance, a number of list scheduling algorithms have been developed for mapping task graphs to homogeneous multiprocessors that have a fully connected topology. In case these fundamental assumptions about the homogeneity or topology of the multiprocessor network change, the algorithms are no longer directly applicable and must be modified. In practice, since heuristics have been tuned to a specific problem, modifications are hard to perform. It is often the case that the heuristic loses much of its computational efficiency when such modifications are imposed. This reduces the viability of using heuristics in an automated scheduling framework.

Randomized Methods

As opposed to heuristic methods, randomized algorithms offer the advantages of increased extensibility and a wider search of the solution space. An important class of randomized algorithms are simulated annealing based algorithms. Simulated Annealing is an optimization meta-algorithm that is used to find the optimum of some objective over large design spaces. It was first introduced by Kirkpatrick et al. [Kirkpatrick *et al.*, 1983]. The algorithm starts with an initial state in the design space, and initially allows for near-random transitions to "nearby" states. As the annealing

proceeds, these transitions become more restrictive: it becomes more likely that a transitions will only occur to a state that improves the objective. The ability to transition to states that increase the value of the optimization objective allows the algorithm to escape local optima, making these algorithms superior to heuristics. At the same time, the annealing schedule allows the system to settle down as the annealing proceeds. Simulated Annealing has been successfully used for solving scheduling problems with different objectives and constraints [Devadas and Newton, 1989] [Koch, 1995] [Orsila *et al.*, 2008]. Indeed, the ability of simulated annealing based algorithms to flexibly accommodate varied resource and implementation constraints is a key advantage of such methods. A key drawback to simulated annealing methods is that they do not, in practice, provide any guarantees on the optimality of the solution obtained. As a mitigating factor, they do possess a number of parameters that can be tuned to obtain a high quality of solutions within a reasonable runtime.

Exact Methods

Although randomized algorithms can be tuned to provide for a good balance between computational efficiency and quality of solutions produced, it is useful to have a technique that can produce provably optimal solutions to optimization problems. Such exact methods generally use a variant of branch-and-bound to search the solution space. Branch-and-bound methods work by systematically enumerating the solution space and pruning out large subsets of inferior solutions at once using upper and lower bounds to the optimal solution. There have been a number of approaches that use branch-and-bound techniques to solve multiprocessor scheduling problems [Kasahara and Narita, 1984] [Fujita *et al.*, 2003]. Standard solvers such as Mixed Integer Linear Programming (MILP) solvers and Boolean Satisfiability (SAT) solvers can be used to abstract the branch-and-bound search. The optimization problem is then encoded as a set of constraints expressed in a format that is specific to the solver. Such solvers have been used to solve many optimization problems in the operations research community [Atamtürk and Savelsbergh, 2005]. Various MILP and SAT formulations have been proposed to scheduling problems [Bender, 1996] [Thiele, 1995] [Tompkins, 2003]. The advantage of using exact methods is that they can handle additional restrictions to a scheduling problem through the addition of side constraints to the problem. The nature of constraints that can be added is only limited by the constraint structure that the solver accepts. In addition to MILP and SAT, there are many other solvers that handle constraints in various "theories" [Dutertre and de Moura, 2006] [Wang *et al.*, 2006]. The main disadvantage of exact methods is that the computation cost can be significant. In many cases, exact methods can only handle problems

that have a very small solution space. Thus they may not be applicable to practical problems of larger size.

1.4 Application of compile-time methods to practical scheduling problems

The key to successful application deployment on single-chip multiprocessors lies in effectively mapping the concurrency in the application to the architectural resources provided by the platform. Compile-time scheduling is an essential step in this mapping process. In this dissertation, we shall investigate and evaluate the use of heuristics, randomized algorithms and exact algorithms for scheduling concurrent tasks onto multiprocessor platforms. Scheduling problems that arise in realistic application deployment and design space exploration frameworks can encompass a variety of objectives and constraints and require different features to be exposed in the application and architecture models. In order for scheduling techniques to be useful for realistic exploration frameworks, they must then be sufficiently flexible to be applied to a range of problems. They must also produce high quality solutions and must be computationally efficient.

We shall compare different scheduling methods in the context of three scheduling problems that can arise in practical mapping scenarios. First, we consider the problem of allocating tasks with dependencies to processors and scheduling them in time in order to minimize the end-to-end completion time (or makespan) of an application. The inputs to the problem are the task graph, representing the tasks and dependencies, and an architecture model that contains the number of processors and the interconnection topology of the processors. We assume that we have complete knowledge of the tasks and performance characteristics of tasks at compile-time. As an example, we consider the mapping of two realistic applications, namely IPv4 packet forwarding and Motion-JPEG encoding on Xilinx FPGA based soft multiprocessors [Xilinx Inc., 2004]. We compare three scheduling algorithms for solving the resulting mapping problem: a list scheduling heuristic, a simulated annealing randomized method and a constraint optimization based exact method. This is presented in Chapter 2 of the dissertation.

Second, we focus on the problem of mapping applications with large data sets onto a system consisting of a CPU and GPU connected with a PCI-Express bus. We focus on a different facet of mapping concurrent applications, that of resource allocation of data to GPU and CPU memory and scheduling the data transfers between them in order to optimally utilize the communication

bandwidth between the CPU and GPU. The application is again represented as a task graph with nodes representation tasks and edges representing dependence and data transfers between tasks. In addition, we also explicitly denote the sizes of the data transferred between the tasks. For this problem, we assume that all tasks are data-parallel and will be mapped onto the GPU. However, all the intermediate data that needs to be stored during the application execution may not fit into GPU memory. In such cases, some of the data needs to be spilled to CPU memory. Such spills are slow (needing to go over the PCI-Express bus) and can become a bottleneck for the system. The total amount of data that is spilled must, therefore, be minimized. It can be shown that the order in which tasks and data transfers are scheduled can have a significant impact on the amount of data spills. Thus the optimization problem is to find the start time of tasks as well as the timing of data transfers between tasks in order to minimize the total amount of data spills. In Chapter 3, we evaluate the use of different scheduling methods to solve this optimization.

In the third problem, we revisit our first problem of task allocation and scheduling to minimize makespan in the context of applications with variable execution times and task dependencies. Such variations often arise due to data-dependent executions or jitter in memory access times in the application. We propose an extension to the compile-time models and optimization frameworks to handle statistical variations in application execution times and task dependencies. We capture task dependencies, task execution and communication times with statistical variables rather than as constant values. We then propose statistical optimization methods that utilize statistical analysis to evaluate the makespan of different schedules. The optimization objective is to minimize a given percentile of the application makespan distribution to provide a guaranteed Quality of Service (QoS) for the application. As an example, a IPv4 packet forwarding algorithm may require that 95% of all packets complete within a specified period. In such a case, the finish time of the application is best measured as the 95^{th} percentile of the makespan distribution. In Chapter 4, we define statistical models to take variability in execution times into account and a statistical analysis procedure to compute the application finish times for different schedules. In Chapters 5 and 6, we extend the scheduling algorithms used in Chapter 2 to find schedules that minimize application finish time.

We can see that the above scheduling problems, while all arising from practical mapping concerns, require different models, have differing constraints and optimize for different objectives. A programmer may need to select a particular model and objective based on the specific concerns posed by the design that is being evaluated. In such a context, a general tool for design space exploration must be capable of handling these varied constraints and objectives. The focus of this dissertation is to provide guidance in choosing scheduling methods that best trade-off the flexibility

with solution time and quality of the resulting schedule.

1.5 Contributions of the dissertation

This dissertation has two main contributions. First, we investigate and evaluate a range of techniques to solve three scheduling problems: task allocation and scheduling onto multiprocessors, resource allocation and scheduling data transfers between CPU and GPU memories, and scheduling applications with variable task execution times and dependencies onto multiprocessors. We develop a code framework that we use to evaluate different scheduling techniques for these task-level scheduling problems. The scheduling techniques that we investigate and evaluate comprise heuristic methods, simulated annealing based optimization and constraint optimization based exact techniques. These techniques have different trade-offs with respect to the quality of solution, computational efficiency and extensibility. The system programmer can then choose among these available techniques based on the requirements of the problem. Over the course of this evaluation, we show that simulated annealing is a good technique for solving scheduling problems. For the three scheduling problems that we consider in this dissertation, simulated annealing offers solutions that are comparable to other scheduling approaches. At the same time, it offers immense flexibility in being able to accommodate a variety of problem constraints and objectives. Further, the method is scalable and works on large scheduling instances. These three advantages makes simulated annealing an appealing choice for scheduling problems with large and complicated design spaces.

Next, we show that we can extend compile-time techniques to scheduling instances with variability in application execution times and task dependencies. One of the key drawbacks of compile-time scheduling methods has been that they assume exact knowledge of the performance model of the application at compile time. However, real execution and communication times can vary significantly across different runs due to the presence of data dependencies or jitter in memory access times. Static methods typically assume worst case behavior for performance models. However, a large class of soft real-time applications do not require worst case performance guarantees but only require statistical guarantees on steady-state behavior. For such applications, scheduling for worst-case behavior may not best utilize system resources. This motivates our move to statistical models that expose the variability present in the application. In this dissertation, we present scheduling methods that accompany these statistical models for effectively mapping soft real-time applications.

Chapter 2

Static Task Allocation and Scheduling

One of the main challenges in utilizing modern many-core architectures is in programming them to effectively utilize the available parallelism and achieve high-performance implementations. In order to address this challenge, we outlined a Y-chart approach to application deployment on parallel architectures in Chapter 1. In this chapter, we describe the Y-chart approach in the context of mapping streaming applications to soft multiprocessor systems on FPGAs. The central problem to be solved is the issue of allocating and scheduling parallel tasks to multiprocessors to optimize the end-to-end finish time of the application. We discuss static models to model the application and architecture, and then discuss different methods to solve this mapping problem.

2.1 Mapping streaming applications onto soft multiprocessor systems on FPGAs

Streaming applications are characterized by the presence of concurrent computational kernels that process large sequences of data. Such systems have become prevalent in a number of application domains starting from small embedded systems, Digital Signal Processors (DSPs) to large-scale internet routers and cellular base stations. It has been estimated that over 90% of the computing cycles on consumer machines are utilized by streaming applications [Rixner *et al.*, 1998].

Stream-based applications offer extensive opportunities for concurrent implementations on parallel platforms. The elements in the input sequence are typically independent of each other, resulting in data-level parallelism across independent input streams. The computational kernels operating on each piece of data are also typically concurrent; this offers the potential for task-level

parallelism. The challenge then is to map the concurrency in such applications to multiprocessor architecture so as to best utilize architecture resources. The optimization criterion is either application throughput, the aggregate rate at which input data items are processed, or latency, the time taken to process a single input data item.

We now offer two examples of streaming applications, IPv4 packet forwarding and Motion-JPEG decoding. The first example is a networking application, where the stream consists of different network packets that can be concurrently processed. The second is a multimedia application consisting of a sequence of video frames that must be decoded.

2.1.1 Examples of Streaming Applications

IPv4 packet forwarding

The IPv4 packet forwarding application runs at the core of network routers and forwards packets to their final destinations [Baker, 1995]. For each input packet, the forwarder decides the IP address of the next hop and the egress port where the packet should be routed.

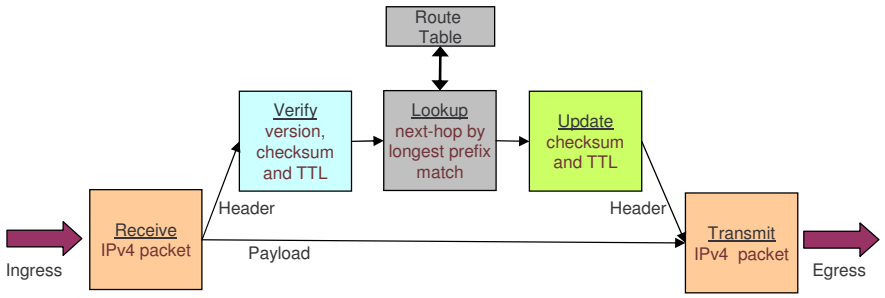


Figure 2.1: Block diagram of the data plane of the IPv4 packet forwarding application.

Figure 2.1 shows a block diagram of the data plane of the IPv4 packet forwarding application. The data plane of the application involves three operations: (a) receive the packet and check its validity by examining the checksum, header length, and IP version, (b) lookup the next hop and egress port by performing a longest prefix match lookup in the route table using the destination address, and (c) update header checksum and time-to-live fields (TTL), recombine header with payload, and forward the packet on the appropriate port. These operations can be classified into operations on the packet header and the packet payload. No computations are performed on the packet payload, which must only be buffered and transmitted. Since all processing occurs on the

header, the performance of the router is determined by the header processing.

Motion-JPEG encoding

Motion JPEG is a video compression standard that encodes a video as a sequence of JPEG images without any inter-frame compression. This standard is commonly implemented in consumer and security cameras.

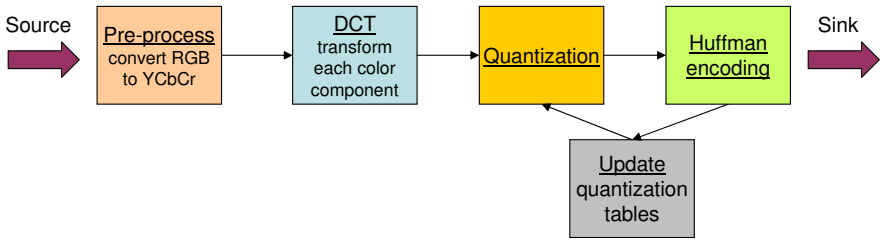


Figure 2.2: Block diagram of the Motion JPEG video encoding application.

Figure 2.2 shows the block diagram of an Motion JPEG encoder. The encoder takes in a sequence of images in raw RGB format. Each pixel in the image has data corresponding to the three color components of red, green and blue. The key kernels of the application are: (a) pre-processing, that converts the color space from RGB to YCbCr format that uses the luminance (Y), blue chrominance (Cb) and red chrominance (Cr) to represent a pixel, (b) a forward integer DCT transform that converts the spatial YCbCr data to the frequency domain, (c) lossy quantization to compress the frequency domain values by dividing each component by a user-supplied coefficient from a quantization table, and (d) a Huffman encoding step that uses run-length compression followed by a lookup into a pre-defined Huffman table to compactly represent the quantized data, and (e) a final step used to update quantization tables based on the differences between the actual and desired compression rates.

Motion JPEG offers parallelism since different kernels can process different data images simultaneously. The performance goal for Motion JPEG encoding is to minimize the end-to-end latency of encoding a given set of JPEG images into Motion JPEG video.

2.1.2 The Mapping and Scheduling Problem

The first step in mapping streaming applications onto multiprocessors is to obtain a description of the parallel tasks and dependencies in the application in the form of an application model.

One way to obtain this is from a natural description of the application in a Domain Specific Language. DSLs provide component libraries and computation and communication models to express the concurrency. Examples of such languages include Click [Kohler *et al.*, 2000] for networking applications, Simulink [The MathWorks Inc., 2005] for dataflow applications, LabVIEW [National Instruments Inc.,] for data-acquisition and processing applications and so on.

Dataflow representations have been commonly used to describe streaming applications in many domain specific languages. For instance, Simulink [The MathWorks Inc., 2005] is a DSL that captures dynamic dataflow in digital signal processing applications. The G-language in LabVIEW [National Instruments Inc.,] also supports static dataflow to describe the processing of data after it is acquired. Software synthesis frameworks have also been built to directly design static dataflow (SDF) applications. The Streamit language [Thies *et al.*, 2002] essentially allows the expression of static dataflow constructs for streaming applications. The Dataflow Interchange Format of [Hsu *et al.*, 2005] is also an example that allows a DSP application to be built up from pre-existing kernels connected using a static dataflow representation. In this work, we use a static dataflow representation to express the parallel tasks and their dependencies.

Since we are interested in optimizing the execution time of the application, the tasks in the dataflow model need to be annotated with a timing-related performance model. This is obtained by profiling each task on a single target processor either in simulation or by actually running it on the target platform. The dependence edges also need to be annotated with the time taken to transfer the data that is communicated on the edge over different hardware links.

Given an annotated dataflow description of the parallel tasks and dependencies and a particular multiprocessor architecture, solving the mapping problem involves the following steps: (1) allocating the tasks in the dataflow to different processing elements and (2) compute a start time for each task (respecting task execution times and dependence constraints) to minimize the completion time of the application. The performance objective may also be to maximize the throughput of the application. The mapping is subject to a number of resource constraints. For instance, the topology of the architecture may impose restrictions on the allocation of dependent tasks. Other common constraints include data memory limits and instruction store limits.

2.1.3 Soft Multiprocessor Systems on FPGAs

A target platform that allows us to explore a range of architectures and demonstrate the value of our mapping techniques is a Field-Programmable Gate Array (FPGA). FPGAs are highly cus-

tomizable and can be used as a fabric to instantiate a variety of multiprocessor architectures. A *soft multiprocessor system* is a multiprocessor network that is created out of processing elements, memories and interconnection structures on an FPGA [Jin *et al.*, 2005] [Ravindran *et al.*, 2005]. Soft multiprocessors offer the opportunity to try different numbers of processors, interconnect schemes, memories and peripherals to perform an effective design space exploration for the target application. Soft multiprocessors can also help open the world of FPGAs to software programmers if they are used as end deployment platforms.

Recognizing this opportunity, leading FPGA vendors such as Xilinx and Altera have offered tools and design flows to help design soft multiprocessor networks on FPGAs. Xilinx offers the Embedded Development Kit (EDK) [Xilinx Inc., 2004] that helps instantiate networks of IBM PowerPC cores on chip and soft processors (called the MicroBlaze processor) connected to on-chip BlockRAM memory and off-chip SDRAM memory. Each soft processor has local instruction and data memory and is connected to it through a Local Memory Bus (LMB). The processors are connected to each other either directly through high-speed point-to-point FIFO links called Fast Simplex Links (FSLs) or through shared on-chip memory using an On-chip Peripheral Bus (OPB). Altera offers similar capabilities through their System-on-Programmable Chip (SOPC) builder using Arm and Nios processors [Altera Inc., 2003].

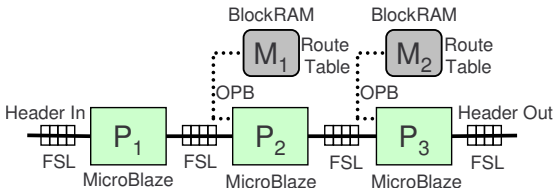


Figure 2.3: Architecture model of a soft multiprocessor composed of an array of 3 processors.

Figure 2.3 shows an example of a soft-multiprocessor network for IPv4 packet forwarding instantiated on a Xilinx FPGA. The figure shows a pipeline of three MicroBlaze processors. The MicroBlaze processor is a 32-bit RISC processor with configurable instruction and data memory sizes. The pipeline is set up using point-to-point FSL links. Each of these MicroBlaze processors will execute one or more of the primary kernels in the IPv4 forwarding application. The second and third MicroBlaze processors in the pipeline are connected to on-chip BlockRAM containing the packet forwarding route table using the On-chip Peripheral Bus.

We note that the topology of the architecture in Figure 2.3 is not fully-connected. This architec-

ture therefore imposes constraints on the mapping problem described in Section 2.1.2. In particular, it is not possible to pass any data from the third processors in the pipeline to the first or second; hence tasks that require input data from the third processor also need to be assigned to the same processor. Further, constraints can also arise because of memory connectivity: any tasks requiring access to the route table cannot be allocated to the first processor.

2.1.4 The need for automated mapping

One of the key benefits of using a soft multiprocessor network on FPGAs is the ability to evaluate different architectures. The resource-constrained mapping problem needs to be solved for each design to evaluate the performance of the application. Manual evaluation of each of these designs can be cumbersome, especially for designs with a large number of processors and irregular architectures. Today, a commercially available FPGA can support more than 20 processors with complex memories and interconnection schemes. For such large designs, there is a need for an automated framework that can estimate the performance of the application based on a model of the architecture. In the next section, we show such an automated mapping step in the context of streaming applications.

2.2 Automated Task Allocation and Scheduling

In this section, we describe an automated mapping step that statically evaluates the performance of a concurrent application on a multiprocessor architecture. Such a step is key to efficiently explore complex design spaces.

We first develop models for the concurrent application and the multiprocessor architecture that allow for automated performance evaluation. We then formally describe the mapping problem and present it as an optimization problem.

2.2.1 Static Models

Static models are used to capture the knowledge about the concurrency in the application, task execution times and parallelism in the architecture at compile time. Such models are used in static scheduling, where scheduling decisions related to allocation and ordering of tasks are done at compile time. In contrast, dynamic scheduling methods make such decisions at run time. Dynamic scheduling does not assume knowledge about the models at compile time, but instead attempts to

optimize application performance on-the-fly with minimum overhead.

In the context of design space exploration for soft multiprocessors, dynamic evaluation of designs would necessitate the synthesis of each possible multiprocessor design. Synthesis and placement of large soft multiprocessor networks on FPGAs can take many hours, belying the requirement of quick design evaluations. Static techniques can quickly prune a large part of the design space without the need for expensive synthesis [Gries, 2004].

Static techniques are also useful when mapping is aimed at application deployment onto a fixed architecture and not design space exploration. For many streaming applications, the dataflow of the application is known at compile time. In some streaming application domains such as Digital Signal Processing, a static dataflow graph is in fact a natural representation of the application [Lee and Messerschmitt, 1987a]. When the application workload and concurrency in the application are known at compile time, static scheduling techniques are viable. In such contexts, static scheduling avoids the overhead associated with dynamic scheduling.

Application Model

A popular representation of the concurrency in the application is the task graph model [Graham *et al.*, 1979] [Bokhari, 1981] [Gajski and Peir, 1985] [Kwok and Ahmad, 1999b]. The task graph model is a directed acyclic graph $G = (V, E)$, where V is the set of vertexes representing the tasks, and $E \subseteq V \times V$ is the set of edges representing task dependencies and communication. A task represents a sequence of instructions that are executed on a single processor without preemption. Task dependencies restrict a task to start only after all its predecessors are complete. Figures 2.4 and 2.5 show the task graph for the IPv4 packet forwarding application and the Motion JPEG application. Each of the tasks represents a kernel that must operate on the input data stream.

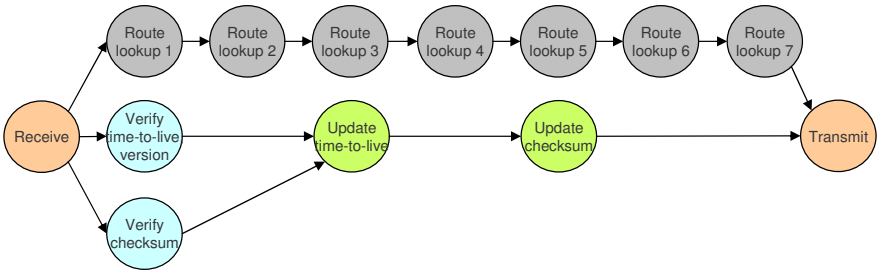


Figure 2.4: Task graph the IPv4 header processing application.

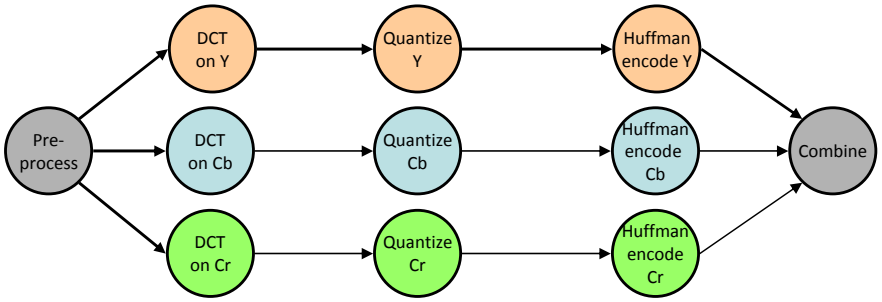


Figure 2.5: Task graph for the Motion JPEG video encoding application.

The task graph is a variant of a dataflow representation of the application. In particular, it represents a acyclic homogeneous static dataflow graph. A homogeneous data flow graph is one where each task is run once for a single application input. An acyclic task graph does not have any cyclic dependencies between tasks, which means that there is a valid topological order for tasks to be scheduled.

Static dataflow representations have been commonly used in describing streaming applications. However, such static dataflow graphs are not naturally homogeneous or acyclic. Lee and Messerschmitt [Lee and Messerschmitt, 1987a] present a scheme to convert general SDF graphs to acyclic homogeneous static dataflow graphs. The conversion is based on unrolling the original graph for a certain number of iterations. The temporal dependencies between the iterations is depicted by adding extra edges to preserve the data dependencies. The number of iterations to be unrolled is the maximum number of inputs. In practice, the input stream may be infinite or very large, resulting in unmanageably large task graphs. We then typically restrict the number of iterations to a constant. We shall discuss the implications of the choice of this constant as we discuss the results of our approach in Section 2.4.4.

Architecture Model

We model the architecture as a set of processors P connected by an interconnection network $C \subseteq P \times P$. The interconnection network represents point-to-point links between pairs of processors. This is a natural fit for the point-to-point FIFO links present in soft multiprocessor systems. In case the architecture has a bus connecting two or more processors through shared memory, the corresponding architecture model will have those processors completely connected with all sets of possible edges. In this case, the model works as an abstraction of the communication between the

processors.

This model does not have an explicit representation of memory. The assumption in this model is that every processor has local memory associated with it. If shared memory is used for communicating between the processors, it is modeled as a communication edge. This model is a accurate fit for distributed multiprocessor systems such as soft multiprocessors on FPGAs. In such contexts, the typical use case is that data communication between processors is done via the point-to-point links, and all memory blocks are either local to a processor or are read-only if shared between processors. The importance of memory in such systems arises due to either memory size constraints or the contribution of memory access cost to task execution times. Both of these are accounted for in our model, either when modeling performance or as constraints on the mapping problem.

Performance Model

A performance model is necessary to estimate the throughput or latency of a particular mapping. We associate each task v in the task graph with weights $w(v, p)$, indicating the execution time of task v on a given processor p. Note that this allows the processors to be heterogeneous. Each edge in the task graph (v_1, v_2) is associated with a weight $c((v_1, v_2), (p_1, p_2))$, which indicates the cost of data transfer from v_1 to v_2 when such transfer occurs over the communication edge between processors p_1 and p_2. A transfer would occur over the link (p_1, p_2) only if v_1 and v_2 were allocated to processors p_1 and p_2 respectively. The edge (p_1, p_2) must be present in C, the list of communication edges. It is common to assume that the cost of local communication within a processor is negligible, i.e. $c(e, (p_1, p_1)) = 0$. Figure 2.6 shows the annotated task graphs for IPv4 packet forwarding and Motion JPEG encoding. For our soft multiprocessor network, all the processors and communication channels are identical, consisting of MicroBlazes and point-to-point FIFO links respectively. Hence a single value is sufficient to represent the execution time of each task and communication delay along each edge.

This model is commonly used in multiprocessor scheduling problems. The model is popularly called the *delay model* in scheduling literature. In the delay model, the values of w and c are considered to be real-valued constants. In practice, the worst-case execution time or average-case execution time of tasks is typically used. Such estimates may be available through profiling or by analytical modeling of each task.

The delay model abstracts the complexity of each task and only exposes a single performance number for the task. It groups local and shared memory access time inside a task with the compu-

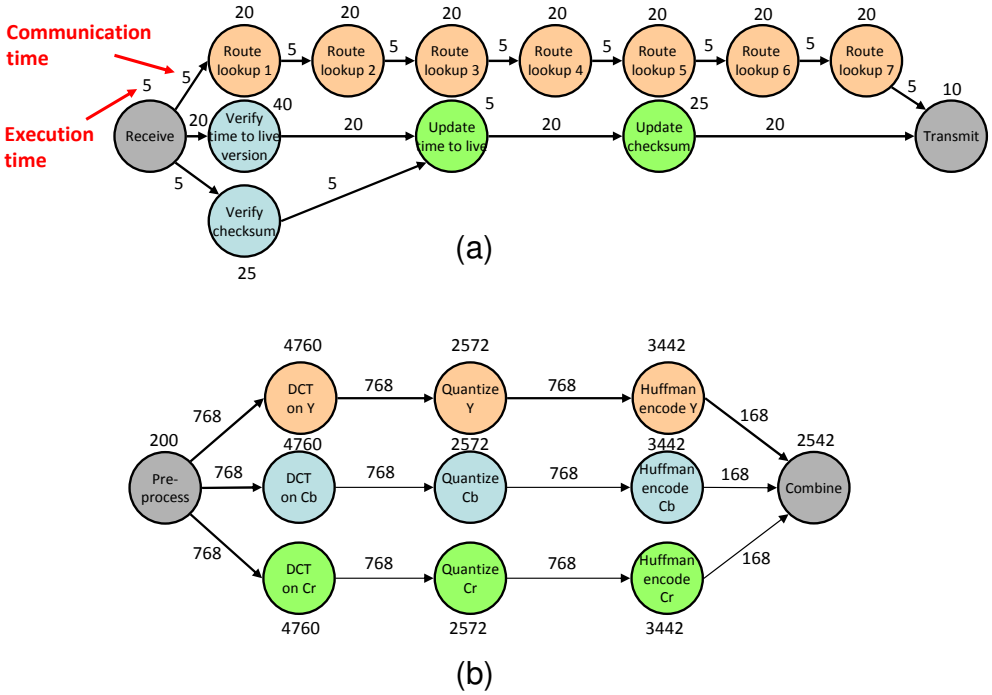


Figure 2.6: Execution and communication time annotations for the task graphs of the (a) IPv4 forwarding and (b) Motion JPEG encoding applications.

tation time for the task. It does not take into account any variability in memory access or execution time. However, it exposes the key components required for reliable performance estimates: the parallel tasks, dependencies and execution and communication times. The simplicity of the model makes it easier to analyze and optimize the timing performance of the application. The delay model would be insufficient if we wished to optimize for some other metric such as power, area, amount of memory transferred or cost. Such problems would require the model to be annotated with the appropriate metrics. We present one example of this in Chapter 3.

2.2.2 Optimization Problem

Given the application and architecture models, the optimization problem is to find a valid allocation of tasks to processing elements and a valid schedule for these tasks to maximize throughput or minimize latency. We now give a mathematical formulation of the task allocation and scheduling problem. We start with defining a valid allocation and schedule and then describe the optimization problem.

Valid Allocation and Schedule

Given a task graph $G = (V, E)$, and an architecture model (P, C), a valid allocation A is a function $A : V \rightarrow P$ that maps each task in V to a processor in P. In the presence of a restricted processor interconnect topology, the following constraint must be satisfied:

$$(v_1, v_2) \in E \Rightarrow (A(v_1), A(v_2)) \in C$$

which indicates that all edges that represent data communications between tasks must be mapped onto a valid communication edge in the architecture.

Given a performance model (w, c) and a valid allocation A, a valid schedule S is a function $S : V \rightarrow \Re^+$ that provides a start time for every task. This must satisfy the following constraints:

$$\begin{aligned}
&\forall (v_1, v_2) \in E \text{ (dependence constraints)},\\
(a)\ &S(v_2) \geq S(v_1) + w(v_1) + c((v_1, v_2), (A(v_1), A(v_2)))\\
&\forall v_1, v_2 \in V, v_1 \neq v_2 \text{ (ordering constraints)},\\
(b)\ &A(v_1) = A(v_2) \Rightarrow S(v_1) \geq S(v_2) + w(v_2) \vee S(v_2) \geq S(v_1) + w(v_1)
\end{aligned}$$

Constraint (a) enforces the dependence constraints between tasks. If there is a dependence edge from v_1 to v_2, then v_2 can start only after v_1 completes and finishes communicating its outputs to v_2. The cost of communication $c((v_1, v_2), (A(v_1), A(v_2)))$ is zero if v_1 and v_2 are assigned to the same processor. Constraint (b) orders tasks assigned to the same processor. The constraint specifies that if v_1 and v_2 are assigned to the same processor, either v_1 finishes before v_2 starts or v_2 finishes before v_1 starts. This constraint is responsible for ensuring that tasks executions are not preempted, and that at any point of time a single processor executes at most one task.

Constraint (b) is redundant if there is a dependence edge between v_1 and v_2. In that case, constraint (a) captures the necessary relation between the start times of $v_!$ and v_2. Given a task graph $G = (V, E)$, we call tasks v_1 and v_2 independent if $(v_1, v_2) \notin E^T$, where E^T is the set of edges in the transitive closure of $G = (V, E)$. It is possible to capture constraints between independent tasks as additional edges in the graph. We define

$$E'' = \{(v_1, v_2) \notin E^T | A(v_1) = A(v_2) \wedge S(v_2) \geq S(v_1) + W(v_1)\}$$

to be the set of *ordering* edges that define the total order in which tasks allocated to the same processor execute. The dependence and ordering edges define the constraints that must be satisfied when deciding the start time of tasks.

In addition to the above constraints, there could be other constraints imposed by the architecture. Potential constraints could include:

- Memory size: The memory requirements for individual tasks must be met by the processors to which they are allocated. This restricts the space of allocations.
- Preferred allocation: A task may be required to be assigned to a specific processor. This can arise when tasks require access to memory or I/O that are only connected to certain processors.
- Clustering: A set of tasks must be assigned to the same processor. This constraint occurs when tasks share a large amount of state, and communication costs between processors are prohibitive. While these considerations are taken into account during the mapping, an explicit clustering constraint can help simplify the scheduling problem to be solved.

Such constraints should also be respected while solving the mapping problem.

The constraints described in this section attempts to capture the relevant restrictions imposed by practical scheduling problems. However, the model on which these constraints are based does have certain limitations. We list a few of them below.

- The granularity of tasks is fixed in our model. In general, we might want a hierarchical representation of tasks and an automated way to pick the right granularity to best load balance the tasks.
- The model does not capture variability in task execution times. In general, data-dependent executions and jitter can cause execution and communication times to vary.
- The model does not explicitly denote shared memory. The presence of shared memory is only indirectly captured through communication links. As such, arbitration constraints and race conditions are not handled.

Inspite of these limitations, the scheduling problem described in this section is a good representative of problems that arises during scheduling. As such, the techniques that we develop over the course of this work should carry over to other scheduling problems.

Optimization Objective

Given a valid allocation and schedule, a common optimization objective is to minimize the latency of the application. This optimization objective has been well studied in scheduling literature

and is referred to as the *makespan* of the application. The makespan of a valid allocation and schedule is defined as $max_{v \in V} S(v) + w(v, A(v))$. After adding the ordering edges to the graph, the makespan can simply be computed as the longest path in the graph. The makespan gives us a measure of the parallel speedup of the application as compared to a sequential implementation. Minimizing the makespan corresponds to finding the allocation and schedule with the maximum parallel speedup.

A related objective which is common for streaming applications is to maximize throughput. We can cast the problem of maximizing throughput to the problem of minimizing the makespan of multiple iterations of a task graph. In Section 2.2.1, we had discussed the unrolling of a general SDF graph to obtain a homogeneous acyclic task graph. This unrolled graph is also useful for computing throughput.

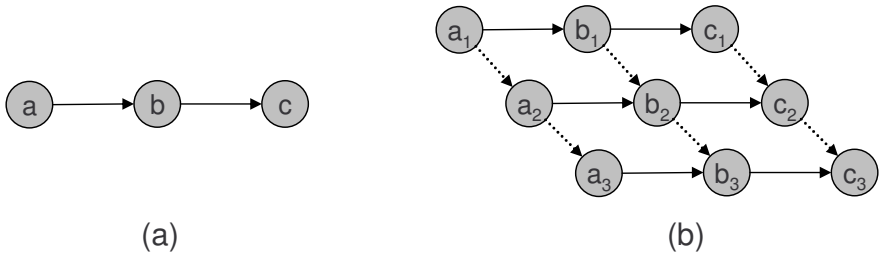


Figure 2.7: Throughput computation in the delay model: the task graph in (a) is unrolled for three iterations in (b).

Given a task graph G, we create a graph G' which contains the tasks in G repeated for a number of iterations. Figure 2.7 shows a graph and its unrolled version. The unrolled graph has additional edges apart from the edges in G. The dotted dependence edges between tasks in different iterations represent the potential temporal dependence between the iterations. If there is a dependence across iterations of the task graph (for instance, each task may carry state across inputs), then these edges ensure that a task does not start processing an input before all previous inputs are processed. These edges are unnecessary if there is no dependence between different iterations.

If graph G is unrolled I times to obtain graph G', the throughput of G is estimated to be proportional to I/M, where M is the makespan of G'. Thus maximizing the throughput of G corresponds to minimizing the makespan of G'. The throughput so computed is only a lower bound on the exact throughput, but is more easily computed given our performance models. The accuracy of the approximation can be improved by increasing the number of iterations for which the task graph is unrolled.

2.3 Techniques for Static Task Allocation and Scheduling

In this section, we develop a toolbox of methods to solve the task allocation and scheduling problem to optimize for makespan. The optimization problem of minimizing the makespan has been well studied in scheduling literature. A variety of techniques have been employed for solving this problem that trade-off the quality of results for solution time [Lee and Messerschmitt, 1987b] [Kwok and Ahmad, 1999b] [El-Rewini *et al.*, 1995]. These range from quick heuristic approaches to exact approaches based on constraint programming. We shall now highlight a popular heuristic method for scheduling and then investigate the use of simulated annealing and constraint optimization [Satish *et al.*, 2007] to solve the task allocation and scheduling problem.

2.3.1 Heuristic Methods

There are two main classes of heuristics that have been used for scheduling problems: techniques based on clustering and those based on list-scheduling. The idea behind clustering algorithms is to merge tasks together into clusters that are then allocated to a single processor. Clusters are typically formed by merging tasks along the critical path of the graph [Kim and Browne, 1988] [Gerasoulis and Yang, 1992]. Clustering techniques have been applied to mapping streaming DSP applications to multiprocessors to maximize throughput [Hoang and Rabaey, 1993].

The most efficient heuristic algorithms for scheduling, however, are based on list-scheduling. The basic structure of a list scheduling algorithm is as follows:

- Compute a global ordering of tasks that reflects the priority of scheduling a task
- While there are tasks left to schedule
  - Select the next task to schedule, based on its priority and whose predecessors have completed execution
  - Allocate the task to a processor that allows the earliest start time for the task, while respecting the validity of the allocation

List scheduling algorithms have been studied since the 1970s [Graham *et al.*, 1979]. Approximation bounds have been proven under specialized conditions [Papadimitriou and Yannakakis, 1988] [Graham *et al.*, 1979].

Different list-scheduling heuristics differ in how they decide the priority metric. A common priority metric involves the *static level* of tasks, which is the largest sum of execution times from

the task to any vertex in the graph. Ties between tasks with the same static level are broken by the number of immediate successors that the task has.

A successful variant of list scheduling that has been used for different scheduling problems has been dynamic level scheduling (DLS) [Sih and Lee, 1993]. This algorithm has been shown to yield results close to the optimum in the absence of additional constraints on mapping [Kwok and Ahmad, 1999a] [Davidović and Crainic, 2006] [Koch, 1995]. Dynamic list scheduling differs from list scheduling in that the global ordering of tasks is updated after each allocation in the algorithm. The global order can thus select the effects of past allocations to give accurate estimates of start times of tasks on different processors.

Algorithm 2.1 DLS$(G, w, c, P) \rightarrow \ makespan$

// task allocation and start time variables
1 $A(v) = \epsilon, \ S(v) = \epsilon, \ \forall v \in V$
2 **for** $(i = 1 \ldots |V|)$
// choose next task processor pair to schedule
3 $(v, p) =$ DLSDECIDE(A, S, G, w, c, P)
// update schedule based on selection
4 $S(v) = \max\{DA(v, p, A, S), TF(p, A, S)\}, \quad A(v) = p$
5 **return** $\max_{v \in V} S(v) + w(v)$

The algorithm for dynamic list scheduling is outlined in Algorithm 2.1. The inputs to the algorithm are the task graph $G = (V, E)$, the architecture model (P, C) and the performance model (w, c). The output is the heuristically optimized makespan. This algorithm does not enforce additional constraints on the mapping outlined in Section 2.2.2.

The algorithm first initializes the allocation and scheduling variables(line 1). The algorithm executes in $|V|$ steps(line 2). At each step, we maintain the current allocation A and start times S for the tasks assigned in previous steps. The algorithm chooses a single task for allocation and a single processor on which the task is allocated based on a priority metric (line 3). The A and S variables are then updated based on this assignment (line 4). The algorithm terminates when all tasks have been assigned (line 5).

After a scheduling decision has been made in line 3, the algorithm assigns the task to the earliest time that the task can start on the processor chosen. This is computed in line 4 as the maximum of the two parameters:

- $DA(v, p, A, S)$: the earliest time that all predecessor constraints in Section 2.2.2 are met.

- $TF(p, A, S)$: the earliest time that processor p is free after executing all tasks assigned to it so far.

These parameters are computed as:

$$DA(v, p, A, S) = \begin{cases} \infty & : \text{if } v \text{ has already been assigned} \\ \infty & : \exists (v_1, v) \in E, A(v_1) = \epsilon \\ \max\limits_{(v_1,v)\in E} S(v_1) + w(v_1) & : \text{otherwise} \\ \qquad +c\left((v_1, v), (A(v_1), A(v))\right) & \end{cases}$$

$$TF(p, A, S) = \max_{v\in V, A(v)=p} S(v) + w(v).$$

Algorithm 2.2 DLSDECIDE$(A, S, G, w, c, P) \rightarrow (v \in V, p \in P)$

1 **foreach** $(v \in V, p \in P)$
  // compute dynamic level for each task processor pair
2 $DL(v, p) = SL(v) - \max\{DA(v, p, A, S), TF(p, A, S)\}$
  // return task processor pair with highest dynamic level
3 **return** $\arg\max_{v\in V, p\in P} DL(v, p)$

The key step in DLS is the choice of the scheduling decision to make in line 3. This is addressed in Algorithm 2.2. The algorithm computes a priority for every (task, processor) pair (line 1). The priority is computed as the difference between $SL(v)$, the static level of the task and $TF(p, A, S)$, the earliest start time of the task on the processor (line 2). The pair with the highest priority is then returned (line 3). The static level of a node v is the length of the longest path from the node to any node in G. Nodes with high static levels are more likely to be on critical paths and hence are given higher priority. When comparing different processors, processors that allow earlier start times are more desirable. The metric takes both factors into account.

Extensions have been proposed to DLS to deal with processor heterogeneity and irregular multiprocessor architectures. The strategy followed in these works is to modify the priority based metric in these cases. However, extending DLS for particular constraints is non-obvious and must be handled on a case-by-case basis. A more fundamental limitation of heuristic schemes such as DLS is that they take a very local view of the constraints and optimization objective of the mapping problem. It is, in general, possible that a heuristic will discover that there are no valid solutions to the mapping problem after making a set of local decisions. It is in general not possible to ensure that a combination of local scheduling decisions will even yield a valid global solution. This motivates us to look at more global schemes such as simulated annealing that can judge the validity and

optimality of a schedule more holistically. A simulated annealing based algorithm is capable of flexibly supporting many practical constraints such as irregular multiprocessor topologies, resource constraints, preferred task allocations or task clustering.

2.3.2 Simulated Annealing

Simulated Annealing is an optimization meta-algorithm based on an adaption of the Metropolis-Hastings algorithm, a Monte Carlo method to generate sample states of thermodynamic systems. It was first introduced by Kirkpatrick et al. in 1983 [Kirkpatrick *et al.*, 1983].

Simulated Annealing is a global optimization approach that is used to find the optimum of some objective over large design spaces. The inspiration behind simulated annealing came from annealing in metallurgy, a technique involving heating and subsequent slow cooling of metals to allow metal structures to attain their lowest internal energy states. By analogy with this process, simulated annealing starts with an initial state in the design space, and initially allows for near-random transitions to "nearby" states. This simulates the behavior of metals when heated to high temperatures. As time proceeds, the transitions become more restricted to states that improve the optimization criteria. The restriction of the transitions is defined by a control parameter, called the temperature. As temperature decreases, it is more likely that only transitions that improve the objective be allowed.

The general structure of the simulated annealing algorithm is shown in Algorithm 2.3. The inputs are the initial state s_0, an initial temperature t_0 and a final temperature t_∞. The output is a final state s_{best}. The algorithm initially evaluates the objective function for the initial state (line 1) using the COST function. The algorithm proceeds in a sequence of iterations (line 2). At each iteration i, the control parameter called TEMP(i) is evaluated (line 3). The temperature is a decreasing function of the iteration number. At each iteration, the state and cost at the end of the previous iteration is recalled (line 4). The algorithm then generates a new state s_{new} which is obtained by means of a MOVE function. The cost of the new state s_{new} is then evaluated to c_{new} (line 5). The algorithm then computes the difference in cost between the current and new states (line 6). The proposed transition is accepted if the cost of the new state decreases (we assume a minimization problem). If the cost increases, then the decision to accept or reject the transition depends on the extent of cost increase and the control temperature parameter. At high temperatures, almost all transitions are accepted, while at lower temperatures, only transitions that do not significantly increase the cost are accepted. If a transition is accepted, the current state and cost are updated

(line 7) and the best seen state is updated if necessary (line 8). These iterations continue until the temperature drops to below t_∞ (line 9). The best state seen is then returned (line 10).

Algorithm 2.3 SIMULATED_ANNEALING$(s_0, t_0, t_\infty) \rightarrow s_{best}$

```
1   s_best = s_o, c_0 = c_best = COST(s_o)
2   for (i = 0 ... ∞)
      // set temperature for iteration i, generate new move and evaluate cost
3     T = TEMP(i)
4     s_i = s_{i-1}, c_i = c_{i-1}
5     s_new = MOVE(s_i), c_new = COST(s_new)
      // decide whether to accept or reject the move
5     Δc = c_new − c_i
6     if (Δc < 0 or Prob(Δc, T) ≥ Rand(0, 1))
        // accept transition
7       s_i = s_new, c_i = c_new
8       if (c_i < c_best) s_best = s_i, c_best = c_i
9     if (t_i < t_∞) break
10  return s_best
```

A common criticism of randomized algorithms such as simulated annealing is that there is no guarantee that the algorithm will stabilize to an optimal or near-optimal solution given any amount of execution time. Further, the exploration of the design space is said to be ad-hoc and not systematic. However, we contend that for many practical scheduling instances, simulated annealing algorithms yield very good results. The widespread use of simulated annealing algorithms in many optimization problems including scheduling [Devadas and Newton, 1989] [Koch, 1995], [Orsila *et al.*, 2008], endorses our stand. Further, for problems with large and complex design spaces, it may not be possible to systematically explore the entire design space in a reasonable amount of time. In such a case, a simulated annealing algorithm that works by sampling points in the design space and avoids local minima can frequently yield better solutions than a systematic approach.

Simulated Annealing has been successfully used for solving scheduling problems. Devadas and Newton [Devadas and Newton, 1989] use simulated annealing to solve scheduling problems for datapath synthesis. Koch [Koch, 1995] advances a technique for performing multiprocessor allocation and scheduling on DSP platforms. A key advantage to using simulated annealing is that the technique can flexibly accommodate varied resource and implementation constraints.

Orsila et al. [Orsila *et al.*, 2008] present a survey of simulated annealing algorithms for the multiprocessor scheduling problem. Different simulated annealing algorithms vary in how they tune various functions and parameters in the general algorithm shown in Algorithm 2.3. These

include the COST, TEMP, PROB and MOVE functions and the initial and final temperatures. We use the techniques in [Koch, 1995] and [Orsila *et al.*, 2006] to tune these characteristics.

We now describe the key simulated annealing characteristics used in solving the allocation and scheduling problem.

- COST function: The cost function COST(s) specifies the value of the optimization objective for a given state s. In the context of the scheduling and allocation problem, the set of valid states is the set of valid allocations and schedules in Section 2.2.2. The COST function is naturally expressed as the makespan of the valid allocation and schedule.

- TEMP function: The temperature function describes the temperature as a function of the iteration number. Common temperature functions include geometric and fractional schedules [Orsila *et al.*, 2008]. In this work, we use the function described by Koch [Koch, 1995].

$$Temp(i) = \begin{cases} \frac{Temp(i-1)}{1+\delta\frac{Temp(i-1)}{\sigma_{i-L,i}}} & \text{if } (i \bmod L) = 0 \\ Temp(i-1) & \text{otherwise} \end{cases}$$

  where

$$\sigma_{i-L,i} = \text{stddev}\{\text{COST}(s_k) | i - L \leq k \leq i \}$$

  According to this function, the temperature only changes once every L steps. The extent of the temperature decrease is determined by two factors: δ, which is a constant multiplicative factor (set to 0.3 in our implementation), and $\sigma_{i-L,i}$, which accounts for the standard deviation in the makespans produced in the last L steps. A high standard deviation means that the annealing has not stabilized as yet; hence the temperature is only slightly decreased. When stabilization occurs, the temperature drops to near zero and the annealing procedure ends.

- PROB function: The probability function determines whether a cost-increasing transition is accepted or not. A normal probability function uses an increasing function of the $-\frac{\Delta c}{Temp}$ ratio. The function we use is:

$$Prob(\Delta c, Temp) = exp(-\frac{\Delta c}{Temp})$$

  This function leads to lower acceptance probabilities as either the cost difference Δc rises, or as the temperature $Temp$ gets lower.

- MOVE function: The move function defines the "neighborhood" on which the transitions occur. The MOVE function is based on a random move of one or more tasks from one processor

to another to create a new valid schedule from the current one. In our work, we choose exactly one task for movement. Let s be the current state characterized by an allocation A and a schedule S. Let $V_p = \{v \in V | A(v) = p\}$ be the set of tasks assigned to processor p. As discussed in Section 2.2.2, schedule S imposes a total ordering on these tasks. The MOVE function picks a random task $v \in V$ and a processor $p \in P$ and fixes the allocation $A(v) = p$. It then selects a position in the global ordering of tasks in V_p and schedules the task to run in that position. The start times S are then recomputed for tasks in V_p. We need to ensure that the resulting schedule is valid. This can be assured by checking that there are no predecessors of v among the tasks in V_p after the position chosen, and that there are no successors of v in the tasks in V_p before the position chosen.

- t_0 and t_∞: An annealing procedure must start with a high initial probability of acceptance of transitions p_0 and a low final acceptance rate p_f. We should choose the initial and final temperatures to achieve these probabilities. Using the PROB function above, the temperature t for a fixed probability p is:

$$t = \frac{\Delta c}{\ln(\frac{1}{p})}$$

Assuming that we expect a minimum cost change of Δc_{min} and a maximum of Δc_{max}, the initial and final temperatures are set to:

$$t_\infty = \frac{\Delta c_{min}}{\ln(\frac{1}{p})} < \frac{\Delta c_{max}}{\ln(\frac{1}{p})} = t_0$$

Additional Constraints

A big advantage of simulated annealing is that additional constraints such as processor topologies, memory sizes or task clusterings can be easily imposed on the scheduling problem. The only SA function that needs to change is the MOVE function. In general, the MOVE function can be changed to have a final post-transition check of the validity of the new schedule. If the new schedule obtained after the move is invalid, then the transition is unrolled, and a new random move is taken. The overhead of imposing additional constraints is the small cost of performing the validity check and the unrolling of the schedule if required. This makes simulated annealing a very appealing optimization method to solve complex and heavily constrained optimization problems.

Although simulated annealing algorithms provide for a good balance of flexibility, efficiency and quality of solution produced, it is still useful to find the optimal solution to a scheduling problem, if it is possible to do so in a reasonable amount of time. Even if an optimal solution cannot be

obtained, it is frequently useful to find bounds to the optimal solution that can help evaluate how much improvement can be made to a given solution. Scheduling approaches that focus on finding optimal solutions or bounds to the optimization problem fall under the category of exact approaches.

2.3.3 Constraint Optimization Methods

A commonly used exact approach to solve many optimization problems in operations research is Mixed Integer Linear Programming (MILP) [Atamtürk and Savelsbergh, 2005]. This is because of the ease of expressing constraints and the availability of high performance solvers like the ILOG CPLEX [ILOG Inc.,] solver. MILP based approaches have been tried in various scheduling problems. Bender describes MILP formulations for resource-constrained scheduling problems [Bender, 1996]. Thiele uses MILP to solve complex scheduling problems in memories and buses [Thiele, 1995]. However, MILP methods have been shown to be unsuccessful in scheduling large-scale problems [Davidović *et al.*, 2004] [Tompkins, 2003]. Davare et al. present a taxonomy of possible MILP approaches to the task allocation and scheduling problem [Davare *et al.*, 2006] for solving small-scale problems of up to 30 tasks (or up to a thousand variables and constraints in the problem formulation).

Constraint programming (CP) is another solver method to solve discrete optimization problems. The variables in such methods take discrete values from finite domains and the constraints are usually first-order logic constraints. Ekelin and Jonsson propose a CP approach to solving embedded scheduling problems using the SICtus Prolog constraint solver [Ekelin and Jonsson, 2000]. However, as in MILP techniques, applications of CP techniques to scheduling problems has also been limited to small scale problems.

Since it is difficult to solve the entire scheduling problem in a single pass, a natural solution technique is to attempt to split up the problem into more manageable parts. We now describe one such approach: a decomposition-based constraint programming scheme that uses a iterative combination of a constraint based "master" problem and a graph-theoretic based "sub" problem. The master problem solves a simplified version of the full optimization problem. The sub-problem then analyzes these solutions and in turn learns new constraints to prune portions of the search space. These constraints are then added to the master problem and a new iteration is performed.

Decomposition methods have been used to solve complex optimization problems in operations research. Jain and Grossmann [Jain and Grossmann, 2001] use a decomposition based technique to schedule independent tasks with release and finish times on a multiprocessor. The master problem

is formulated as an MILP problem and the sub-problem as a CP problem. Benini et al. consider a similar multiprocessor scheduling problem, and apply a similar MILP/CP decomposition technique to solving the problem [Benini *et al.*, 2005].

The particular strategy we use is inspired by *Bender's decomposition* [Benders, 1962]. Benders decomposition has been used as a solution strategy for constraint programming and other combinatorial optimization approaches such as Boolean satisfiability [Hooker and Ottosson, 1999] [Hooker and Yan, 1995]. Bender's technique is to start with a simple problem with a small subset of the constraints in the problem and iteratively solve more complex problems by adding back some of the constraints. The choice of which constraints to add is decided by another sub-problem. This choice of constraints to add is key to the efficiency of the approach. It is important to prune as large a search space as possible with the constraints added at each iteration so that not many iterations are required.

For the problem formulation in Section 2.2.2 , we propose a decomposition technique using a combination of a Boolean satisfiability based master problem and a graph-theoretical sub-problem algorithm. The inputs to the algorithm are the task graph $G = (V, E)$, the architecture model (P, C) and the performance model (w, c). In this section, we assume that the processor network is homogeneous (the scheme, however, is also applicable to other constrained networks). . Then the execution time w is only a function of a task and not its allocation. The communication time c is also a function of only the edge (v_1, v_2) in the task graph and not the allocation of tasks v_1 and v_2. The outputs of the algorithm are a valid allocation A and a valid schedule S, as defined in Section 2.2.2.

The decomposition-based approach (DA) algorithm in shown in Algorithm 2.4. The constraints for the master problem are formulated in conjunctive normal form (line 3). The approach consists of a number of iterations (line 4). Each iteration solves the master problem using a satisfiability solver (line 5). A satisfiable solution allocates tasks to processors and orders all tasks belonging to the same processor. The sub-problem then adds the *ordering edges* corresponding to the total ordering of tasks to each processor as described in Section 2.2.2 (line 7). There are two possible cases: the new graph with ordering edges may contain a cycle or may be acyclic. If it does contain a cycle, then the ordering is invalid; hence a set of constraints are added to the master problem that eliminate the cycle (line 9). If the graph is acyclic, then the ordering is valid; the sub-problem then finds the makespan of the schedule as the longest path in the graph and conditionally updates the best makespan found so far (line 11). The sub-problem also adds a set of constraints that attempt to prune parts of the solution space that are guaranteed to have strictly inferior solutions

than the best makespan (line 12). The constraints that are generated by the sub-problem take effect when the next master problem iteration is solved. The process continues until the master problem is unsatisfiable, at which point we have found the optimal makespan.

Algorithm 2.4 DA$(G, w, c, P, C) \rightarrow\ makespan$

```
1   makespan = ∞
2   φ = empty CNF formula
3   BASECONSTRAINTS(G, w, c, P, C, φ)
4   while (true)
      // master problem
5     x_SAT = SATSOLVE(φ)
6     if (x_SAT = UNSAT)     return makespan
      // sub-problem
7     G' = UPDATEGRAPH(G, x_SAT)
8     if (G' contains a cycle)
9       CYCLECONSTRAINTS(G', x_SAT, φ)
10    else
11      makespan = min{makespan, MAKESPAN(G') }
12      PATHCONSTRAINTS(G', w, c, makespan, x_SAT, φ)
```

We now describe in detail the master and sub-problem formulations.

1. **Master problem:** The master problem uses Boolean variables to encode task allocation and ordering. A subset of the constraints to make the allocation and ordering valid are also added as conjunctive normal form clauses. There are three sets of Boolean variables that are used:

$$\forall v \in V, \forall p \in P,$$

$$x_a(v,p) = \begin{cases} 1 & : \text{ if task } v \text{ assigned to processor } p \\ 0 & : \text{ else} \end{cases}$$

$$\forall (v_1, v_2) \in E,$$

$$x_c(v_1,v_2) = \begin{cases} 1 & : \text{ if tasks } v_1 \text{ and } v_2 \text{ are assigned} \\ & \quad \text{to different processors} \\ 0 & : \text{ else} \end{cases}$$

$$\forall v_1, v_2 \in V, \quad (v_1, v_2) \notin E^T, (v_2, v_1) \notin E^T,$$

$$x_d(v_1,v_2) = \begin{cases} 1 & : \text{ if task } v_1 \text{ precedes task } v_2 \\ 0 & : \text{ else} \end{cases}$$

The x_a variables are used to encode the ordering of tasks to processors. The ordering variables x_d encode the "ordering edges" that are used to order independent tasks assigned to the same processor. The communication edges encode whether there is a communication delay between two dependent tasks (i.e. whether they are assigned to different processors).

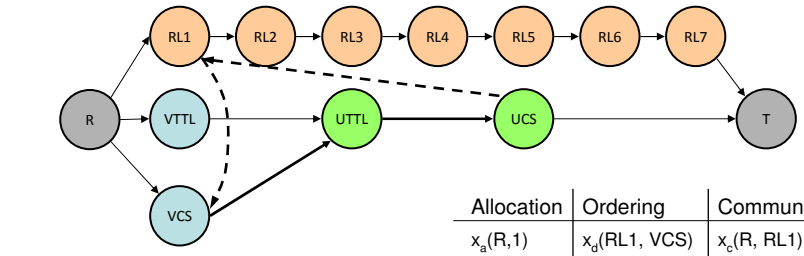
The constraints related to the master problem include:

$$\forall v \in V,$$
$$(A1) \quad \Big(\bigvee_{p \in P} x_a(v,p) \Big)$$
$$\forall v \in V,\ \forall p_1, p_2 \in P, p_1 \neq p_2,$$
$$(A2) \quad x_a(v,p_1) \wedge x_a(v,p_2) \ \Rightarrow\ 0$$
$$\forall (v_1,v_2) \in E,\ \forall (p_1,p_2) \in (P \times P) - C,$$
$$(A3) \quad \big(x_a(v_1,p_1) \wedge x_a(v_2,p_2)\big) \ \vee\ \big(x_a(v_1,p_2) \wedge x_a(v_2,p_1)\big) \ \Rightarrow\ 0$$
$$\forall (v_1,v_2) \in E,\ \forall p_1, p_2 \in P, p_1 \neq p_2,$$
$$(C1) \quad x_a(v_1,p_1) \wedge x_a(v_2,p_2) \ \Rightarrow\ x_c(v_1,v_2)$$
$$\forall v_1, v_2 \in V, (v_1,v_2) \notin E^T, (v_2,v_1) \notin E^T,\ \forall p \in P,$$
$$(D1) \quad x_a(v_1,p) \wedge x_a(v_2,p) \ \Rightarrow\ x_d(v_1,v_2) \vee x_d(v_2,v_1)$$
$$(D2) \quad x_d(v_1,v_2) \ \wedge\ x_d(v_2,v_1) \ \Rightarrow\ 0$$

Constraints $A1$, $A2$ and $A3$ encode constraints on task allocation. Constraint $A1$ specifies that each task must be allocated to at least one processor. Constraint $A2$ specifies that a task cannot be allocated to more than one processor. Constraint $A3$ indicates that dependent tasks must be allocated to processors that can communicate with each other. These three constraints naturally fall out of the discussion in Section 2.2.1. Constraint $C1$ sets a communication variable corresponding to an edge to 1 if the tasks on the edge are allocated to different processors. Constraints $D1$ and $D2$ encode the ordering constraint: if two independent tasks are allocated to the same processor, one of them should be ordered before the other.

2. **Sub problem:** The sub problem is responsible for identifying whether the allocation and schedule returned by the master problem is valid or not, and if valid the makespan of the schedule. In order to do this, we add the "ordering edges" represented by the x_d variables to the task graph. The edges added are defined by:

$$\big\{(v_1,v_2) \mid x_d(v_1,v_2) \wedge \big(\exists p \in P : x_a(v_1,p) \wedge x_a(v_2,p)\big)\big\}$$



Cycle: {RL1, VCS, UTTL, UCS}

Cycle Constraint:
x_d(RL1,VCS) ∧ x_d(UCS, RL1) ⇒ 0

Allocation	Ordering	Communication
x_a(R,1)	x_d(RL1, VCS)	x_c(R, RL1)
x_a(RL1,2)	x_d(UCS, RL1)	x_c(R, VCS)
x_a(RL2,3)		x_c(UTTL,UCS)
x_a(RL3,3)		x_c(VCS,UTTL)
x_a(RL4,3)		x_c(RL1,RL2)
x_a(RL5,3)		x_c(UCS,T)
x_a(RL6,3)		
x_a(RL7,3)		
x_a(VTTL,1)		
x_a(VCS,2)		
x_a(UTTL,1)		
x_a(UCS,2)		
x_a(T,3)		

Figure 2.8: A solution to the master problem that contains a cycle, hence representing an invalid schedule.

As an example, Figure 2.8 shows a master problem solution to scheduling the IPv4 application onto the architecture shown in Figure 2.3. The figure shows the task graph (with names abbreviated) to which the ordering edges (represented as dashed edges) have been added. The x_a, x_c and x_d variables that have been set to 1 are also indicated. After the addition of these edges, the master problem solution has a cycle, highlighted in the figure with edges that are bold. Since all edges have a positive weight, the presence of such a cycle means an invalid schedule, and all solutions that contain this cycle must be eliminated from consideration. In order to do this, we add the constraint $x_d(RL1, VCS) \wedge x_d(UCS, RL1) \implies 0$. In general, given a cycle consisting of edges $E_c = \{(u_1, v_1), (u_2, v_2), \ldots, (u_k, v_k)\}$, the associated cycle constraint is:

$$\Big(\bigwedge_{(u,v) \in E_c - E} x_d(u, v) \Big) \Rightarrow 0 \, .$$

After this cycle has been added, the master problem is solved again and yields a new solution. Figure 2.9 presents this solution. The solutions differ in the orientations of the edges

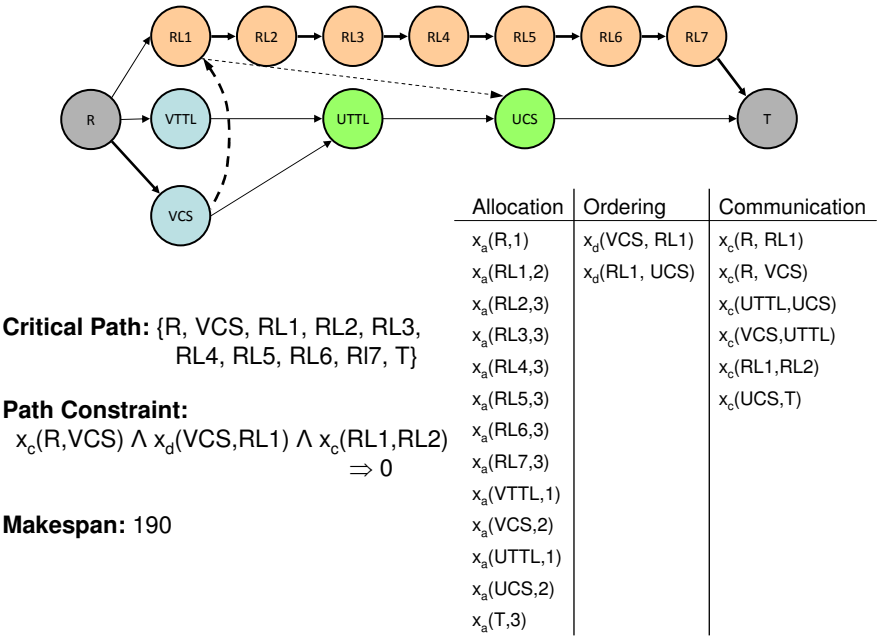


Figure 2.9: A second solution to the master problem that represents a valid schedule.

$(RL1, VCS)$ and $(UCS, RL1)$. Now, the solution has no cycles and is thus valid. The makespan of a valid allocation and schedule can be obtained by a longest path computation on the graph with ordering edges. The longest path in Figure 2.9 has been highlighted with bold edges. The makespan is the length of this longest path, which is 190 in this example. The algorithm keeps track of the best makespan found so far and updates it with the makespan of this schedule, if necessary. Given a schedule with a critical path, it is clear that no other solution that contains this path can have a lower makespan. In this example, this means that it is no longer necessary to consider schedules that contain the path $R \rightarrow VCS \rightarrow RL1 \rightarrow ... \rightarrow RL7 \rightarrow T$. In order to record this information, we add the constraint $x_c(R, VCS) \wedge x_d(VCS, RL1) \wedge x_c(RL1, RL2) \implies 0$. It is to be noted that the only scheduling variables that impact the delay of a path are the set of ordering edges (x_d variables set to 1) and the set of communication edges (x_c variables set to 1). Only such variables need to be recorded.

In general, we could record any path in the graph that has a makespan worse than the best makespan found so far. If $E_p = \{(u_1, v_1), (u_2, v_2), \ldots, (u_k, v_k)\}$ are the edges comprising a path in G' with delay greater than or equal to the best makespan at an iteration, then the

associated path constraint is:

$$\Big(\bigwedge_{(u,v)\in E_p - E} x_d(u,v)\Big) \wedge \Big(\bigwedge_{(u,v)\in E_p \cap E, A(u)\neq A(v)} x_c(u,v)\Big) \Rightarrow 0.$$

We keep adding either cycle or path constraints at each iteration of the sub problem, until there is no solution to the master problem. At this stage, we have considered all possible allocations and schedules and we can stop.

This scheme should eventually yield the optimal makespan. However, the time required to finish computing the optimal makespan can exceed reasonable time limits. In such a case, we can elect to stop the algorithm at any time, and take the best makespan found so far as the result of the algorithm.

In order to improve the performance of the solver, it is easy to change the algorithm to take in a heuristic solution and use that makespan as the initial value in line 1 of Algorithm 2.4. In general, any upper bound to the makespan can be used here. In our experiments, we start with the results of the DLS algorithm. This helps the DA procedure to avoid redoing the work performed by the DLS algorithm.

Additional constraints

The additional constraints in Section 2.2.2 can be added as constraints to the master problem in the DA approach. The sub-problem and the nature of the constraints added by the sub-problem to prune the solution space remain unchanged. One example of such a side constraint is constraint $(A3)$ in the master problem. This constraint enforces processor topology constraints by ensuring that dependent tasks must be allocated to processors that can communicate with each other. If there are other side constraints such as memory size, task clusterings or preferred allocations of tasks, they should be added to the master problem.

2.3.4 Lower bounds

The above algorithms give us valid makespans for the task allocation and scheduling problems. These are upper bounds to the optimal makespan. It is sometimes useful to obtain lower bounds as well. A lower bound to the scheduling problem gives us the lowest possible makespan for any valid allocation and schedule. No scheduling algorithm can give better makespans than the lower bound.

For the task allocation and scheduling problem, there are two categories of lower bounds. The first assumes that there are no task dependencies, and converts the scheduling problem to a load

balancing problem. For a task graph $G = (V, E)$ scheduled on $|P|$ processors, a lower bound to makespan is $\frac{\sum_{v \in V} w(v)}{|P|}$. Thus assumes that tasks are perfectly load balanced across processors.

The other class of lower bounds assumes that task dependencies exist, but that there are no resource restrictions. In particular, it is common to assume that there are an infinite number of processors. The simplest possible bound is then the longest path in graph G, assuming zero communication costs. Fernandez and Bussell [Fernandez and Bussell, 1973] propose an alternate bound that takes into account the minimum possible communication between tasks. This has been implemented by Fujita et al. [Fujita and Nakagawa, 1999].

2.4 Results

In this section, we present the results of our experiments to evaluate different techniques of solving the task allocation and scheduling problem to minimize makespan. We compare the Dynamic List Scheduling (DLS) algorithm to simulated annealing (SA) and the decomposition-based constraint programming approach (DA). We implemented all algorithms on a 2.66 GHz Core 2 Quad system with 2 GB RAM running Linux. The DA approach was implemented on top of the MiniSAT solver [Een and Sörensson, 2003].

2.4.1 Benchmarks

We used two sets of benchmarks. The first was the two streaming applications of IPv4 packet forwarding and Motion JPEG encoding. Performance profiles were collected by implementation on a soft MicroBlaze core. The FPGA platform used was a Xilinx Virtex 2VP50 using the Embedded Development Kit (EDK). The task graphs in Figure 2.6 for IPV4 and Motion JPEG were unrolled 1-10 times to exploit the data-level parallelism. For IPv4 forwarding, the number of copies is the number of inputs ports in the forwarder. For Motion JPEG, the number of copies determines the number of frames that are buffered and can simultaneously be decoded. The second benchmark is a set of random task graphs from Davidović et al. [Davidović and Crainic, 2006]. These are reputed to be among the harder scheduling problems to solve and incorporate a variety of graph structures found in real applications.

2.4.2 Comparisons on regular processor architectures

Tables 2.1 and 2.2 show the makespan results of the Motion JPEG and IPv4 forwarding applications on fully-connected processor architectures with 2,4,6 and 8 processors. Column 1 shows the number of tasks in the application on unrolling the task graph from one to ten times. The other columns show the results of DLS, SA and DA algorithms along with the lower bound to makespan. The DA approach was started with the result of the DLS algorithm, and hence never returns a makespan worse than the DLS makespan. The best makespan obtained by the DA procedure after 5 minutes is taken as the result. Any DA runs that yielded optimal results before 5 minutes are highlighted in bold. We also indicate the lower bounds obtained by the DA procedure in the "LB" column. For all DA instances that are proved to be optimal, the LB is represented by a "-".

# Tasks	# Processors = 2				# Processors = 4			
	DLS	SA	DA	(LB)	DLS	SA	DA	(LB)
13	20198	19328	**19328**	-	14352	14352	**14352**	-
24	36702	36666	36636	34964	22386	21786	**21618**	-
35	54264	54128	54162	52446	30588	31140	30102	26223
46	72438	71600	71884	69928	39148	39460	38826	34964
57	89576	89098	89126	87410	48144	47926	48144	43705
68	107178	106568	106564	104892	56738	57156	56738	52446
79	124190	123860	124190	122374	65482	65776	65422	61187
90	142102	141292	141744	139856	74044	74568	74042	69928
101	159224	158888	159224	157338	82728	82952	82728	78669
112	176590	176472	176590	174820	92366	92060	92366	87410

# Tasks	# Processors = 6				# Processors = 8			
	DLS	SA	DA	(LB)	DLS	SA	DA	(LB)
13	14352	14352	**14352**	-	14352	14352	**14352**	-
24	19112	19112	**19112**	-	19112	19112	**19112**	-
35	24578	25126	24578	22936	23872	23872	**23872**	-
46	30590	31972	30392	27696	28632	28632	**28632**	-
57	37310	37602	35954	32456	33392	33392	**33392**	-
68	42462	43456	41780	37216	38152	40300	**38152**	-
79	47982	48332	47976	41976	42912	44224	**42912**	-
90	54068	54762	53680	46736	47672	50462	**47672**	-
101	59948	62374	59600	52446	52432	54542	**52432**	-
112	66618	67350	65874	58274	57728	60056	**57192**	-

Table 2.1: Makespan results for the DLS, Simulated Annealing (SA), and decomposition approach (DA) on task graphs derived from MJPEG decoding scheduled on 2, 4, 6, and 8 fully connected processors.

Table 2.3 shows the results for the set of random benchmarks from Davidović et al.[Davidović

# Tasks	# Processors = 2				# Processors = 4			
	DLS	SA	DA	(LB)	DLS	SA	DA	(LB)
15	155	155	**155**	-	155	155	**155**	-
28	270	260	260	250	175	175	**175**	-
41	395	385	395	375	225	235	220	195
54	520	510	510	500	290	295	290	250
67	640	635	640	625	350	345	345	313
80	765	760	760	750	415	415	410	375
93	885	885	885	875	495	475	470	438
106	1015	1010	1010	1000	555	545	535	500
119	1155	1135	1135	1125	615	615	600	563
132	1280	1260	1260	1250	670	675	660	625

# Tasks	# Processors = 6				# Processors = 8			
	DLS	SA	DA	(LB)	DLS	SA	DA	(LB)
15	155	155	**155**	-	155	155	**155**	-
28	175	175	**175**	-	175	180	**175**	-
41	200	210	**200**	-	200	215	**200**	-
54	240	255	**225**	-	230	245	**225**	-
67	280	285	270	245	250	280	**245**	-
80	320	330	305	285	290	315	**285**	-
93	365	395	345	325	330	345	**325**	-
106	410	435	385	365	370	400	**365**	-
119	445	470	425	405	410	450	**405**	-
132	470	530	470	445	455	475	**445**	-

Table 2.2: Makespan results for the DLS, MILP, and decomposition approach (DA) on task graphs derived from IPv4 packet forwarding scheduled on 2, 4, 6, and 8 fully connected processors.

and Crainic, 2006]. For each of these examples, the optimal results are known a priori. The table shows the difference between the optimal results and the makespans obtained from each of the algorithms. The results are tabulated for task graphs consisting of 50 and 100 tasks with different edge densities (the percentage of the number of edges in the transitive closure of the task graph to the total number of edges). For each task size and edge density, we obtain the makespan percentage difference from the optimum for the random graph scheduled on 2,4,6,8,9,12 and 16 processors. We then tabulate the average of these seven percentages. For the DA approach, we additionally list the number of graphs (out of the seven) that were solved to optimality.

The above tables show that the DLS heuristic performs very well on regular fully-connected architectures. The technique is usually within 5-10% of the optimum makespan. This is in line with results reported in the literature [Kwok and Ahmad, 1999a][Davidović and Crainic, 2006]. The DA approach is successful in reporting optimal solutions on task graphs with 100-150 tasks. The

# Tasks	Edge Density	DLS	SA	DA (# optimal)	Average $\frac{DLS-SA}{DLS}\%$	Average $\frac{DLS-DA}{DLS}\%$
50	00	4.8	3.1	3.3 (1)	1.6	0.3
50	10	13.5	15.0	3.3 (0)	-1.5	8.9
50	20	16.2	16.1	4.6 (0)	0.1	9.8
50	30	16.0	18.8	4.0 (0)	-2.4	10.3
50	40	15.9	18.7	1.9 (3)	-2.5	12.0
50	50	14.4	20.3	0.4 (5)	-5.4	12.3
50	60	8.8	17.4	0 (7)	-7.7	7.8
50	70	4.8	13.6	0 (7)	-8.4	4.4
50	80	3.2	18.1	0 (7)	-14.6	2.9
50	90	3.0	18.5	0 (7)	-14.3	2.7
Average		10.1	16.0	1.9	-5.5	7.1
# Optimal Solutions		0 / 70	0 / 70	37 / 70	-	-

# Tasks	Edge Density	DLS	SA	DA (# optimal)	Average $\frac{DLS-SA}{DLS}\%$	Average $\frac{DLS-DA}{DLS}\%$
100	00	3.3	1.6	3.2 (0)	1.6	0.1
100	10	25.0	31.0	15.7 (0)	-5.6	7.1
100	20	16.3	24.2	15.7 (0)	-7.8	0.5
100	30	16.0	23.1	14.6 (0)	-6.4	1.2
100	40	8.7	18.2	8.3 (0)	-8.3	0.3
100	50	7.1	17.1	5.9 (0)	-8.9	1.1
100	60	6.5	30.0	4.3 (2)	-13.9	2.1
100	70	4.1	13.7	1.7 (4)	-8.6	2.3
100	80	2.9	16.9	0.8 (5)	-12.8	2.0
100	90	1.1	14.7	0.3 (6)	-12.0	0.8
Average		9.1	20.1	-10.3		1.8
# Optimal Solutions		0 / 50	0 / 50	17 / 70	-	-

Table 2.3: Average percentage difference from optimal solutions for makespan results generated by DLS, SA, and DA for random task graph instances scheduled on 2, 4, 6, 8, 9, 12, and 16 fully connected processors.

simulated annealing algorithm tends to perform somewhat worse than either DLS or DA on such regular architectures. From the tables, we can see that it can be up to 25% worse than the best DA solution.

An interesting observation from Table 2.3 is that the DA technique performs better when the graph has a high density of edges. For such graphs, the ordering between the tasks is almost fully defined by the edges in the graph itself. In this case, we have very few scheduling x_d variables, reducing the number of scheduling decisions we must make. However, the SA technique is more uniform over graph of different edge densities, except for the zero edge density case. When there

are no edges, the scheduling problem boils down to a load balancing problem without precedences, which is a simpler problem. Both SA and DLS perform well on this case. The DLS heuristic has an interesting trend: it performs well at the two extremes of edge density but does worse in between. The heuristic optimality seems to be the best measure of the inherent difficulty of the problem. The scheduling problem is hardest to solve at moderate edge densities.

The other dimension of comparison of our techniques is the runtime of each optimization technique. The DLS heuristic is the fastest approach, completing in under a minute under all cases. The SA approach can take up to three minutes on different instances, with runtimes for individual cases being unpredictable. The DA approach has the longest runtime, and is stopped at five minutes.

From the above discussion, it is clear that the DLS algorithm is the method of choice when mapping algorithms to regular multiprocessor architectures. However, as we have previously mentioned, the drawback of heuristics becomes apparent when additional constraints are added to the mapping problem. We shall next discuss the results of the scheduling approaches on realistic architecture models that are not so regular.

2.4.3 Comparison on realistic architectural models

We now show the makespan results for the IPv4 forwarding on different architectural designs that we explored using soft multiprocessor networks. Figure 2.10(a-c) show the different designs we wish to evaluate. These designs include a pipeline of three processors in (a), replicated array of processors in (b), and an irregular architecture in (c).

Table 2.4 shows the results of makespan optimization for tasks obtained from unrolling the IPv4 task graph from 1 to 10 times on the processor architectures (a) through (c) of Figure 2.10. For each architecture, we show the results of all our optimization approaches. The heuristic DLS approach was modified to support arbitrary processor topologies as proposed in [Bambha and Bhattacharyya, 2002]. We can easily extend the SA and DA approaches to provide support for different topologies as described in Sections 2.3.2 and 2.3.3. The DA results in the table correspond to the best makespans obtained after 5 minutes of execution. The bold entries indicate the solutions that are proved to be optimal by the DA approach before the 5 minute timeout.

From these results, we can see that the DLS algorithm is not as good on restricted topologies as on fully-connected architectures. This is an understandable trend as the DLS heuristic was initially developed for fully connected models. While DLS still is within 5% of SA and DA results on the small architecture, both SA and DA outperform the heuristic by 15-25% on the larger architectures.

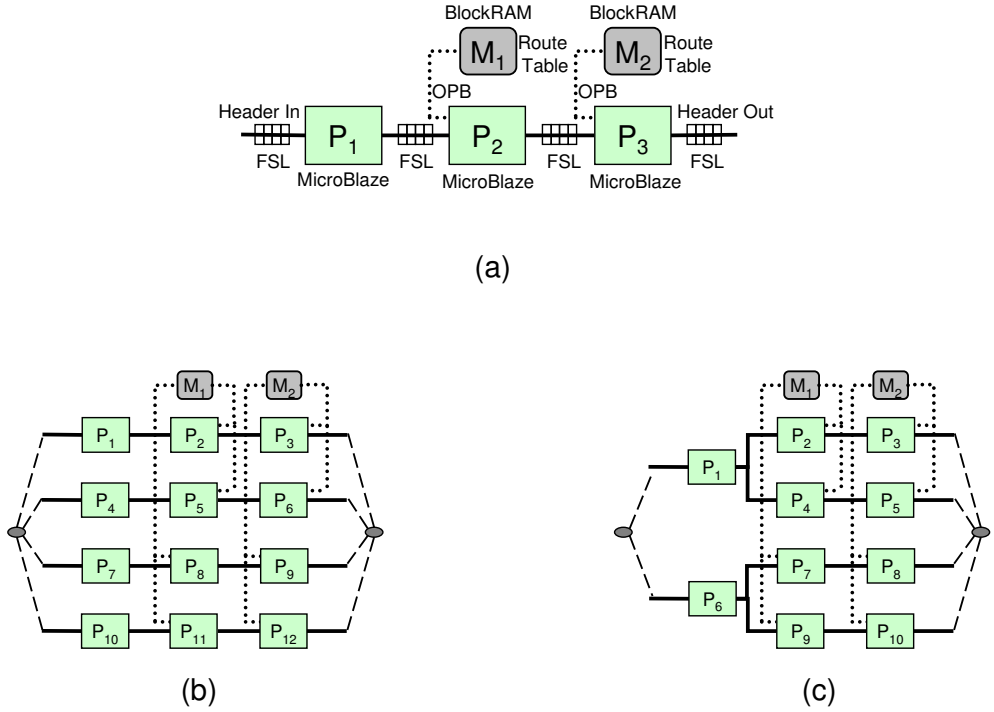


Figure 2.10: Different target architectures for IPv4 packet forwarding.

# Tasks	Arch. (a)				Arch. (b)				Arch. (c)			
	DLS	SA	DA	(LB)	DLS	SA	DA	(LB)	DLS	SA	DA	(LB)
15	185	165	**165**	-	205	165	**165**	-	205	165	**165**	-
28	230	220	**220**	-	205	165	**165**	-	205	170	**170**	-
41	315	305	300	195	205	165	**165**	-	205	170	**170**	-
54	405	385	405	250	205	165	**165**	-	205	170	**170**	-
67	495	470	490	313	250	220	**215**	-	255	215	**215**	-
80	590	555	590	375	250	230	215	155	280	255	240	155
93	700	655	700	438	250	230	215	155	280	255	240	160
106	785	750	785	500	250	245	215	155	280	275	245	182
119	875	835	875	563	340	305	300	174	365	310	310	205
132	985	920	985	625	340	330	300	193	385	355	360	228
Average improvement from DLS (%)	-	5.6	2.1	-	-	12.1	15.8	-	-	12.9	14.9	-

Table 2.4: Makespan results for the DLS, DA, and SA methods on task graphs derived from IPv4 packet forwarding scheduled on the architectures (a) through (c) of Figure 2.10.

For smaller task graphs up to 100 tasks shown in Table 2.4, the decomposition approach yields the best makespans. However, like other exact approaches, the decomposition approach stops scal-

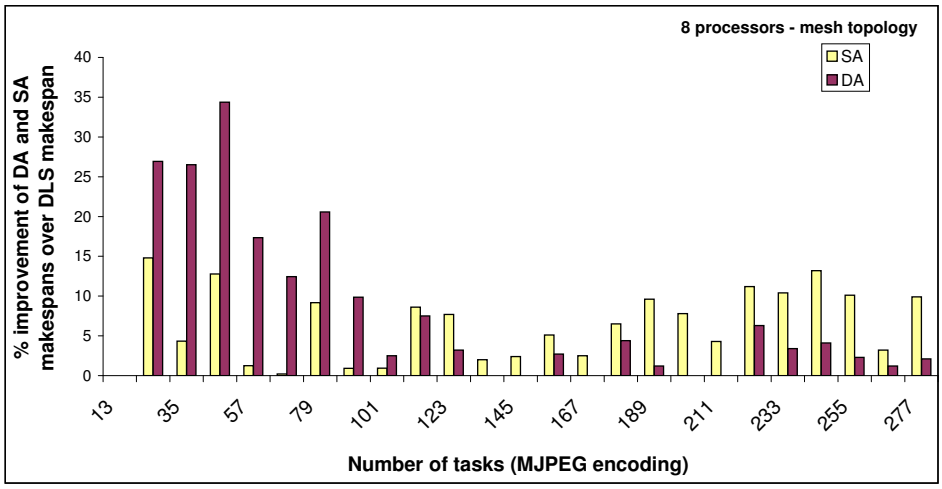


Figure 2.11: Percentage improvements of DA and SA makespans over the DLS makespan for larger task graph instances (with up to 277 tasks) derived from Motion JPEG encoding scheduled on 8 processors arranged in a mesh topology.

ing to problem sizes after a certain point. Figure 2.11 shows the results of our makespan optimization on larger task graphs. We use the Motion JPEG encoding application and vary the number of tasks in the application from 13 to 277 by unrolling the task graph from 1 to 25 times. These graphs are then scheduled on an 8-processor architecture arranged in a mesh topology.

From Figure 2.11, the DA approach stops improving the DLS results significantly if the problem has more than 150 tasks (which corresponds to a few tens of thousands of variables and constraints in the problem formulation). The SA algorithm is more consistent across a wider range of problem sizes: we find that it improves the solution even for problems as large as 500 tasks. For large instances with irregular architectures, SA is the method of choice.

2.4.4 Throughput estimation using makespan

In many streaming applications, the optimization criterion is typically throughput maximization rather than makespan minimization. However, by unrolling the task graph, we can obtain good estimates of the throughput of the mapping from its makespan. As stated earlier, if task graph G is unrolled I times to obtain graph G', the throughput of G is estimated to be proportional to I/M, where M is the makespan of G'. Thus maximizing the throughput of G corresponds to minimizing the makespan of G'. The throughput so computed is only a lower bound on the exact throughput, but accuracy can be improved by increasing the number of iterations for which the task graph is unrolled.

It is, in general, hard to find the exact value of I, the number of times that the graph is to be unrolled. For perfect accuracy, we would have as many iterations as the number of input data. However, for streaming applications, the number of inputs can be very large or even conceptually infinite. In such a case, we will have to use a bounded value for I and accept a lower bound on throughput. However, as we increase the value of I from one, the accuracy of the estimated throughput improves. We illustrate this through an example. Figure 2.12 shows a plot of the estimated throughput for IPv4 packet forwarding on the 3-processor pipeline in Figure 2.12 with increasing values of I. The throughput is measured in Gigabits per second (Gbps), assuming a packet size of 64 bytes and a clock frequency of 100 MHz. All results were obtained by using the decomposition-based approach.

From the figure, we note that as the number of iterations increases, the estimated throughput increases and asymptotically reaches a value of 0.57 Gbps. This value of 0.57 Gbps is also an upper bound on the achievable throughput for this mapping problem. This value of throughput is obtained using a lower bound to the makespan of 90 cycles, obtained when there are no task dependencies. Thus we can see that in this example, as we increase the number of iterations, the estimated lower bound on throughput reaches its highest possible value, which indicates that the estimate of 0.57 Gbps is the exact throughput for this schedule. It also happens that this is the schedule with the optimum throughput.

In general, we find that as we increase the number of iterations, the value of the throughput achieved gradually increases and stabilizes asymptotically to the exact value. As a practical consideration, we do not want to unroll the graph more times as necessary in order to avoid handling large optimization problems. There is thus a tradeoff between the size of the optimization problem and the accuracy of the estimated throughput. For practical scheduling problems, we have found only

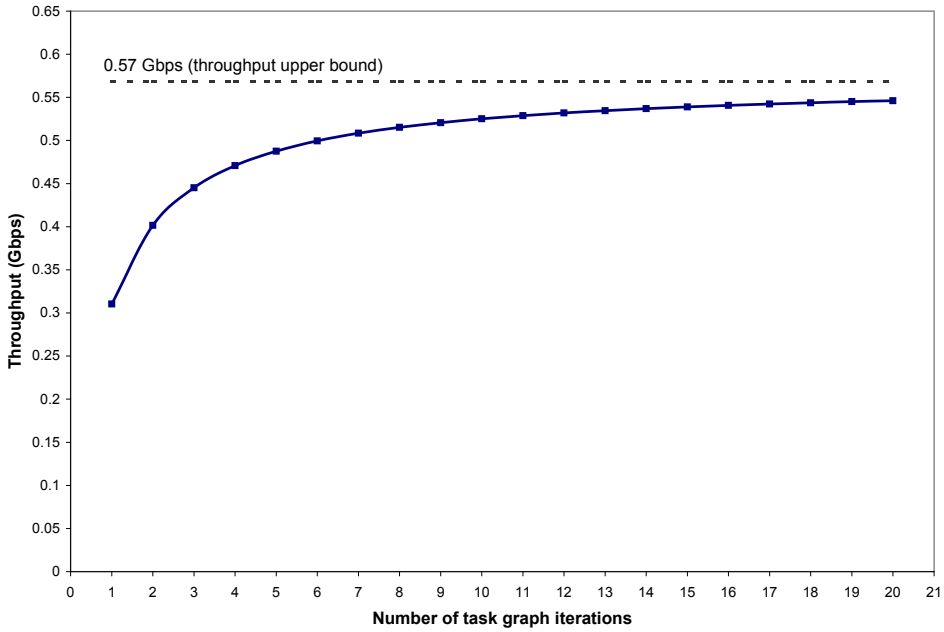


Figure 2.12: Graph of the estimated throughput of the IPv4 packet forwarding application as a function of the number of iterations of the application task graph.

incremental improvements in the accuracy of throughput estimates on unrolling the task graph for more than about 10 iterations.

2.5 Comparing different optimization approaches

The goal of any scheduling approach is to arrive at a high quality schedule while being computationally efficient and being flexible to the incorporation of additional constraints to the mapping. From the results in Section 2.4, it is clear that each of our scheduling approaches has their own advantages and disadvantages with respect to the metrics stated above. At the outset, the DLS heuristic is very effective for the regular multiprocessor models that it is designed to handle. However, the presence of additional constraints can invalidate the assumptions of the heuristic algorithm. The

heuristic then has to be modified for the particular set of constraints under study. Even if the heuristic is so modified, the efficiency of the approach can suffer. We demonstrated this for the problem of mapping applications to architectures with restricted topologies.

The DA approach and other constraint programming approaches have the key advantage that they maintain a high degree of flexibility while also guaranteeing the optimality of the solutions they provide. The range of side constraints that can be added to the problem is only limited by the structure of the constraints that the solver technology can handle. In addition to Boolean clauses in SAT and linear inequalities in an MILP solver, solvers that allow pseudo-boolean constraints, sat-modulo constraints and constraints in many other specialized "theories" have become popular in recent years. With respect to solution quality, it is often the case that optimal solutions cannot be guaranteed in a reasonable completion time of five minutes. Even under such conditions, constraint optimization techniques can provide bounds as to how much the result is inferior to the optimal solution. Constraint optimization techniques also have the useful property that the solutions returned by these techniques are guaranteed to improve with runtime; hence they allow for tradeoffs between runtime and quality of the solution. However, all such exact approaches suffer from the fact that the solution space of scheduling problems is not smooth and as such is difficult to explore completely. Such solvers usually have a "knee" effect: they handle all problems up to a certain size well, but suddenly fail at a certain problem size. As solvers become more powerful, the "knee" will be pushed to larger problem sizes; but the ultimate exponential nature of the runtime of such solvers will persist.

Finally, the use of evolutionary algorithms such as SA provide a middle ground: they are as flexible as any constraint optimization method, but they do not offer the promise of exact solutions. As a consequence, they also do not suffer as much from the NP-completeness of the problem. In our experiments, for large problem sizes of over 150 tasks (which corresponds to a few tens of thousands of variables and constraints in the problem formulation), we found that is most beneficial to use simulated annealing techniques. Simulated Annealing consistently offer improvements over the DLS heuristic for constrained problems.

Chapter 3

Resource allocation and communication scheduling on CPU/GPU systems

In the previous chapter, we described the mapping of streaming applications onto multiprocessor networks for best performance. We formulated this problem as an optimization problem of allocating and scheduling tasks with dependencies onto multiprocessor networks with the objective of minimizing application makespan.

In this chapter, we describe a different problem related to mapping applications with large data sets to a system consisting of a CPU and GPU. This problem illustrates a different facet of mapping concurrent applications, that of resource allocation of data to GPU and CPU memory and scheduling the data transfers between them in order to optimally utilize the communication bandwidth between the CPU and GPU. As in the previous chapter, we will discuss static models and methods to solve the mapping problem for this objective.

3.1 Mapping applications with large data sets onto CPU-GPU systems

CPU-GPU systems have become increasingly popular in recent years as a means of delivering large computational power to the desktop market. Such systems consist of a host CPU with the GPU device connected to the CPU through a PCI-Express link. GPUs support high computational rates (in terms of floating point operations per second) and have a high bandwidth to memory on the GPU board, making such systems ideal for throughput oriented applications.

In order to support high memory bandwidth rates, GPU memory is integrated along with the execution units on the GPU card. A consequence of this is that the memory available on the GPU is

limited and cannot be upgraded by the end-user. Present-day high-end GPUs offer anywhere from 512 MB to 1.5 GB of memory on-board the GPU card. If more than this amount of memory is required by the application, data has to spilled to CPU memory. Such memory transfers go over the PCI-Express bus which has a much lower bandwidth (1-2 Gbps) than the on-board memory bandwidth (over 64 Gbps). This can then become the bottleneck of the system, especially if large amounts of data need to be transferred over the bus. Bleiweiss reports that about half the time in a GPU pathfinding algorithm on large graphs is spent on transferring intermediate data to and from the GPU[Bleiweiss, 2008].

One class of applications that deals with large data sizes are machine learning applications. The intermediate data required to be stored in machine learning applications can easily be much larger than the GPU memory size. It has been reported that the resulting traffic over the PCI-express bus can take well over half the execution time of the application [Sundaram *et al.*, 2009]. For such applications, it is important to try to minimize data transfers.

We explore the issue of minimizing data transfers in the context of applications that are represented as a task graph. We show examples of machine learning applications represented as task graphs in Section 3.1.1. In this context, there are two factors that affect the amount of data transfers required by an application: (1) the order in which different tasks in the application execute (or task scheduling) affects how much intermediate data is live at any point of time and (2) the choice of which intermediate data is spilled to CPU memory as tasks execute (or data transfer scheduling) affects the sizes of the data transferred. The problem of finding the task and data transfer schedules that minimize the total data transfer can be posed as an optimization problem. Sundaram al. [Sundaram *et al.*, 2009] describe a compiler flow that takes a task graph description of the application and generates a task schedule (and corresponding CUDA code) for the application that attempts to minimize memory transfers. However, the approaches used in their work for minimizing data transfers are heuristics that are sub-optimal. This work is an attempt to obtain better solutions to this optimization problem.

We begin by providing the background to this problem. We first describe the machine learning applications that we intend to map onto CPU-GPU systems. We then describe the platform and the nature of the optimization problem that arises during the mapping step.

3.1.1 Applications

Machine learning applications encompass a broad range of applications that are used to extract information from large amounts of data. They are used in various applications such as fields including analysis and processing of signal processing applications, face, gesture and handwriting recognition in image and video processing applications and speech recognition. We show two examples of such applications: an edge detection application used in extracting features for image processing, and a convolutional neural network used in handwriting and face recognition.

Edge detection

Edge detection is used as a pre-processing step in many image recognition applications (such as cancer detection). A classic edge detection technique is the Canny edge detector [Canny, 1986]. Ziou et al. provide a good overview of edge detection techniques [Ziou and Tabbone, 1998]. Edge detection consists of the following steps: (1) a convolution step that takes in an input filter and applies (convolves) it to the input image; different rotations of the kernel are tried out to find edge orientations (2) a threshold and remap step that converts the convolved values to the normal gray-scale value range, and (3) a reduction operation(such as addition/max/absolute value) to combine the remapped results of different edge orientations. Figure 3.1 shows the tasks in the edge detection application. Each of the circles in the figure represents a task. Here, we use four kernel rotations and perform a convolution and remap task for each of them. The values are then reduced pairwise using a max reduction to obtain the final result. Each of these tasks is data-parallel: a single convolution requires the kernel to be applied to every pixel in the image. The remap and max stages also require operations per pixel of the image. The graph in Figure 3.1 is hence a task-level abstraction of the application.

The boxes in the graph denote the data that is transferred between these tasks. The input image is required by every convolution task. Each convolution then produces a different convolved image (which is of similar size to the input image). Each max operation requires two convolved images as inputs. At any point during the execution of the application, many of these convolved images may need to be kept in GPU memory. Each image can be hundreds of megabytes in size at high resolution. All such intermediate data may not fit in GPU memory, leading to spillage to CPU memory.

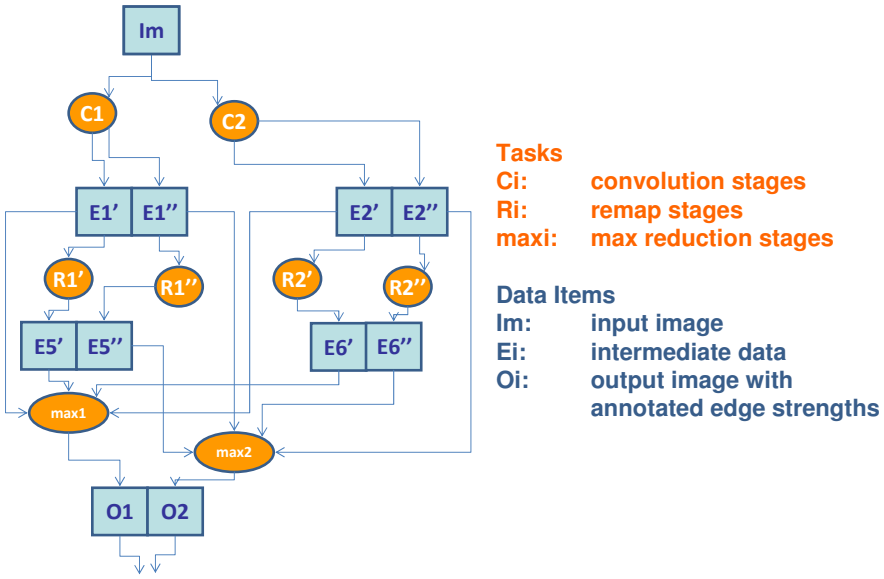


Figure 3.1: Task graph depicting the data parallel tasks and data in the edge detection application.

Convolutional Neural Networks

Convolution Neural Networks (CNNs) are used in many image recognition problems. CNNs are useful in applications like face detection where there is insufficient input data to deal with recognition of features under shifts and rotations [Lawrence *et al.*, 1997]. A typical CNN consists of a set of layers, each of which contains one or more planes. Each layer does one of three operations: a convolution step which is used to extract different features from the image, a sub-sampling step that is used to decrease the importance of the exact position of the feature in the image, and a sigmoidal activation layer that applies a sigmoidal (usually tanh) function to perform the neural network decision. The different planes within each layer are responsible for handling different features in the image. A typical convolutional neural network structure is shown in Figure 3.2. As in edge detection, the circles represent tasks (one per plane in each layer) and the boxes represent data that is transferred between tasks. The structure of the task graph in terms of the number of layers and the types of these layers (convolution, sub-sampling and sigmoidal) can be tuned depending on the exact application the CNN is used in. In our experiments, we used CNNs with 10-15 layers, consisting of a few thousand tasks and data items.

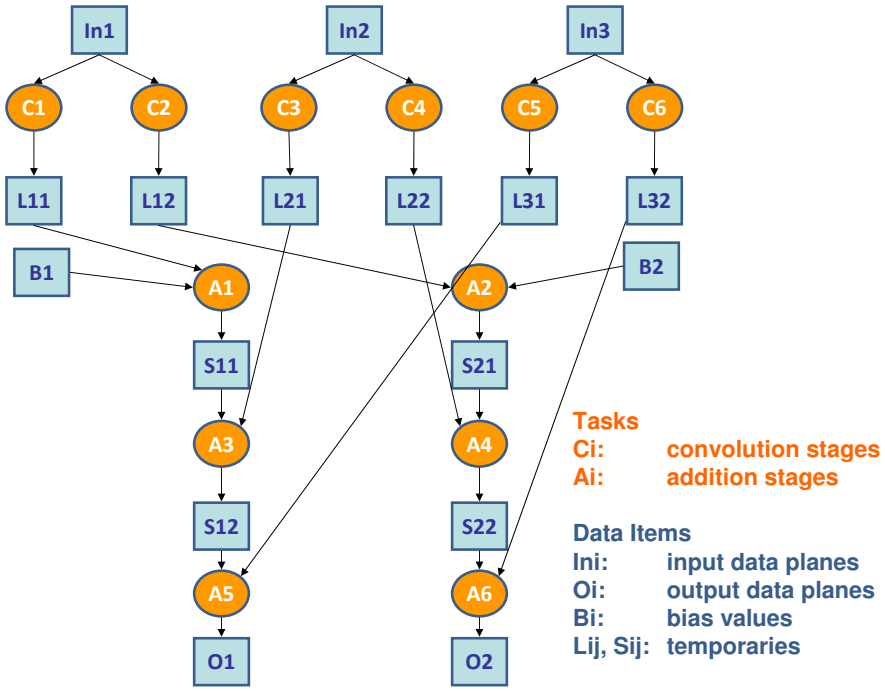


Figure 3.2: Task graph depicting the data parallel tasks and data in a Convolutional Neural Network.

3.1.2 CPU/GPU systems

CPU-GPU systems consist of a host CPU and a Graphics Processing Unit (GPU) card attached to it using a PCI/PCI-Express bus. The function of the GPU in such a system is to accelerate the compute-intensive portions of the application. The CPU is responsible for handling the control dominant parts.

GPU architectures have evolved over the years from graphics-specific functionality to supporting general-purpose computations. However, GPUs still work best on application workloads that mimic typical graphics applications. Such applications tend to be highly data parallel and require high memory throughput while having a minimum of control flow and synchronization. Accordingly, GPUs support massive amounts of data-level parallelism in the application. General purpose programs written for GPUs then need to heavily exploit data-level parallelism to efficiently utilize GPU resources.

Modern programming languages for CPU-GPU systems such as the CUDA language from NVIDIA [NVIDIA Corp., 2007] structure the application as a collection of data-parallel kernels. Each kernel executes on the GPU, and is typically massively multi-threaded. It is usually the case

that only one kernel executes on the GPU at a time; hence each kernel should be capable of saturating the GPU compute resources. The language provides support for expressing the data parallelism within each individual kernel.

The function of the CPU is to handle the high level control flow of the application. In particular, in applications with more than one kernel, the CPU code must decide which data-parallel kernels need to execute on the GPU as also the order in which they execute. The CPU is also responsible for ensuring that all data required by kernels on the GPU are present in GPU memory. This is achieved by explicit transfers to and from CPU memory using pre-defined API calls. Such transfers go over the PCI-Express bus connecting the CPU to the GPU.

3.1.3 The mapping step

The performance of an application executing on a CPU-GPU system is determined by how well each kernel utilizes GPU compute and memory bandwidth resources, as well as how well the CPU code minimizes the memory transfers that are required between the CPU and GPU. While existing GPU languages like CUDA focus on expressing and optimizing each data-parallel kernel, not much emphasis has been placed on the problem of minimizing memory transfers.

One recent development by NVIDIA in the direction of reducing the impact of memory transfers on application execution time has been to allow data transfers into and out of the GPU to occur in parallel with computation using CUDA [NVIDIA Corp., 2007]. However, this comes at the potential cost of having to hold in GPU memory the data that is being transferred at the same time as the data required by the currently executing task. In contrast, if we only allow data transfers to occur in between task executions, we can then perform all data transfers out of the GPU before the task executes, and postpone data transfers into the GPU until after the task completes. In this way, we can free up the maximum possible GPU memory for each task. In this work, we focus our attention on applications with large data sets where memory is a tight constraint, and hence we disallow data transfer to occur in parallel with task execution. We note that the core problem of reorganizing the application to reduce the total amount of data transferred is useful irrespective of whether the data is transferred in parallel with task executions, as this will in any case help reduce the execution time of the application. In this work, we focus on the problem of minimizing data transfers between the CPU and GPU as a proxy for reducing the execution time of applications. We focus our attention on applications with large data sets for which such execution time can be the dominant cost.

The key objective of our work is to help automate the process of minimizing data transfers

between the CPU and GPU. We start with a description of the entire application as a task graph consisting of data-parallel kernels. An example of such a graph is shown in Figure 3.1 for the edge detection application. We assume that only one task executes on the GPU at a given point of time. Each task in the graph consumes one or more input data items (shown as boxes) and generates one or more output data items. A given data item must be stored in memory (and is said to be *live*) from the time it is produced by a task until the time that all tasks that utilize the data finish executing. The order in which the GPU executes the tasks in the task graph determines when each data item is produced and consumed; and hence the set of data items that are live at any point of time.

The total memory footprint of all live data at a given point of time may be greater than the GPU memory size. In such a case, some of the live data needs to be transferred to CPU memory. During the execution of a task, the inputs to the task need to be stored in GPU memory. There should also be enough space available for the outputs of the executing task to be stored. In this work, we assume that the GPU memory is big enough that the inputs and outputs of each single task fits in GPU memory. If this is not the case, then the application cannot be executed on the GPU. We must then either restructure the application by sub-dividing the data into smaller chunks or pick a GPU with larger memory.

All data items other than the inputs of the currently executing task may be transferred to the CPU memory to make space for the inputs and outputs of the task. The data items that are sent to the CPU memory then have to be brought back to the GPU when they, in turn, are required as the inputs to a task. The choice of which data items to transfer out depends on two factors: (1) the size of the data item(s) and (2) how soon the transferred data will be required by another task. It is obviously beneficial to avoid transferring large data items when the GPU memory constraint can be met by transferring smaller data items. It is also beneficial to transfer data items that will not be immediately required as inputs to other tasks, so that GPU memory resources are freed up for the maximum possible amount of time. There might be trade-offs involved between these two factors.

The impact of task ordering on the amount of live data can be significant. Figures 3.3 and 3.4 (taken from [Sundaram *et al.*, 2009]) shows two schedules for the edge detection application. The schedule in Figure 3.3 executes the tasks in a breadth-first order ($C_1 \rightarrow C_2 \rightarrow R'_1 \rightarrow R''_1 \rightarrow R'_2 \rightarrow R''_2 \rightarrow max_1 \rightarrow max_2$). The schedule results in the creation of a lot of live data; indeed there exist three different points in the execution where 7 or more data items (of a total of 11) are live. The schedule in Figure 3.4, on the other hand, attempts to use data items produced by tasks as soon as possible. This schedule only has one point with 7 live data items. It is clear that schedule (b) will require fewer transfers than schedule (a). For the example data sizes and total memory size shown in

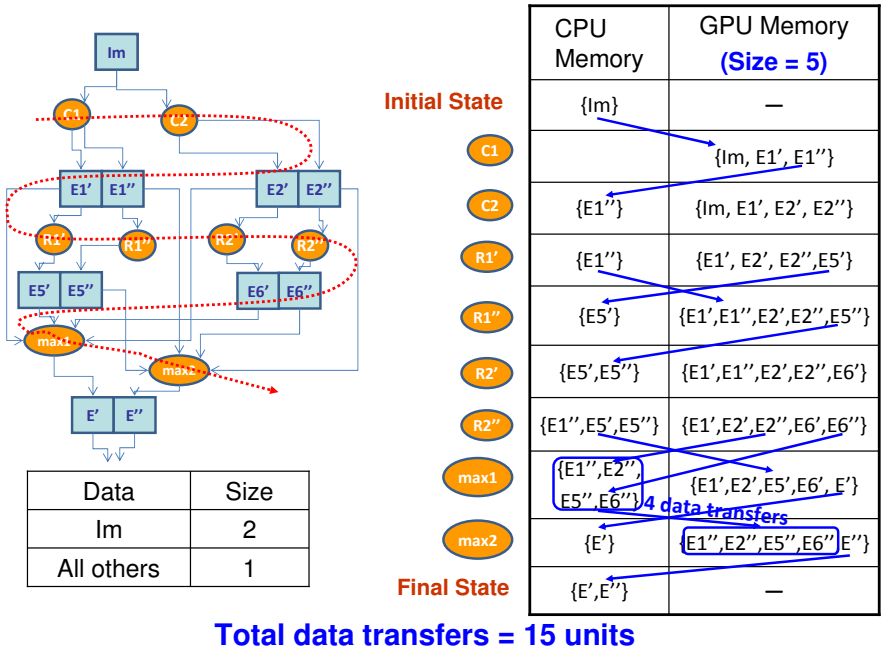


Figure 3.3: A schedule of task and data transfers for the task graph in Figure 3.1.

Figures 3.3 and 3.4 (the real data and memory sizes for edge detection are reported in Section 3.4), it turns out that the two schedules require 15 and 8 units of data transfer respectively. The data transfers for the two schedules differ by nearly a factor of two. The details of the data transfers are shown in the figures.

Given a particular task order, we still have to make decisions about which data items need to be kept in CPU memory and which should be shifted to GPU memory. In the schedule shown in Figure 3.3, the following sequence of operations occur:

1. Initially, the input data Im is present in CPU memory. It must be transferred to the GPU to begin execution. This is a compulsory transfer.

2. Immediately after task C_1 completes (row 2 in the table of Figure 3.3), the input data Im to task C_1 and the outputs E_1' and E_1'' are present in GPU memory. All this data fits in GPU memory.

3. Immediately before executing task C_2 (next on our task order), we must ensure that there is enough space in GPU memory to store the inputs and outputs of task C_2. With the current contents of GPU memory, there will not be enough space to store output data items E_2' and

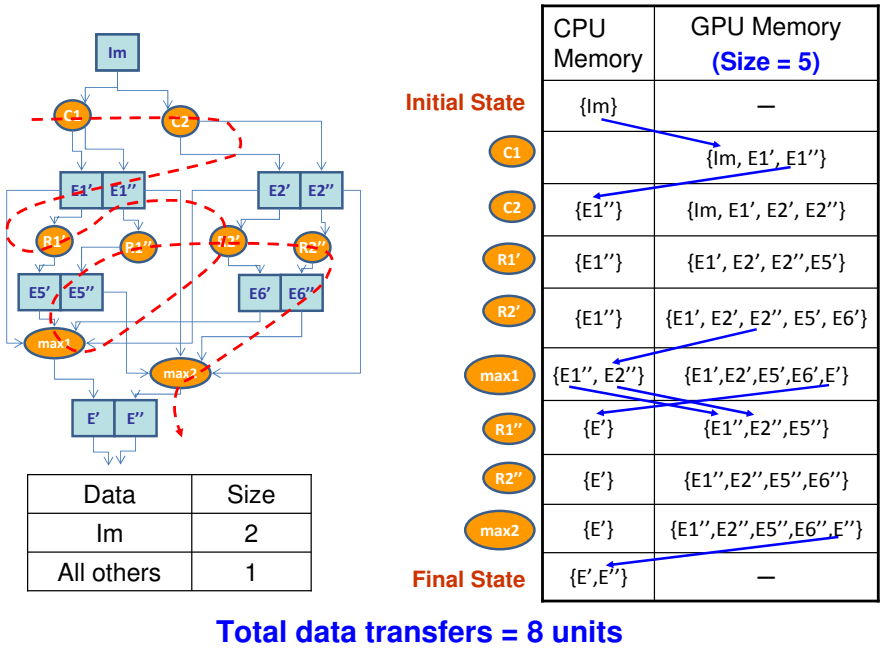


Figure 3.4: A second schedule of task and data transfers for the task graph in Figure 3.1.

E_2''. We must thus transfer one item of data to CPU memory. We could choose to transfer one of E_1' or E_1'', which are live at the point of execution of C_2, but are not used in the execution of C_2. These are both of the same size and thus will incur the same transfer cost. The correct choice turns out to be to transfer E_1'', since this data is used at a later point of execution than data E_1'. In general, such decisions are not easy to make: we prove this problem is in itself NP-complete in Section 3.3.4. After transferring out E_1'' and executing task C_2, the contents of GPU memory are as in row 3 of the table. At this point, we find that data Im is no longer required to be kept in memory: hence it is not live any more. It is then freed up in GPU memory.

4. We continue executing tasks and transferring data to make room for inputs and outputs of executing tasks until we get all our primary outputs. We need the primary outputs of the program in CPU memory, and we have additional compulsory transfers.

As we can see from the edge detection schedules, the problem of minimizing data transfers reduces to a problem of (1) ordering task executions to minimize the data that is live at any time so as to reduce the need for transfers, and (2) scheduling the right set of data items to transfer

out of the GPU (if required) so as to free up GPU memory for the maximum possible period of time. In Chapter 2, we identified generalized techniques such as simulated annealing and constraint optimization that can be used to solve different scheduling problems. We shall explore the use of these techniques to solve the data transfer minimization problem in the rest of the chapter.

3.2 Task and Data transfer scheduling to minimize data transfers

In this section, we formally state and analyze the optimization problem of scheduling tasks and data transfers in order to minimize the total amount of data transferred between the CPU and GPU.

3.2.1 Static Models

The static models that we describe in this section capture the tasks and the data dependencies between tasks in the application, the memory footprint of each data item and the total GPU memory size. We formalize these notions below.

Application Model

The application model is a directed acyclic graph $G = (V, E)$ (hereafter called the task graph), where the nodes V represent either tasks or data items and E represents the data dependencies between the tasks. We can write $V = V_T \cup V_D$, where V_T represents the set of tasks and V_D represents the set of data. These tasks consume input data and produce output data on execution. We denote $E = E_{TD} \cup E_{DT}$, where $E_{TD} \subseteq V_T \times V_D$ denotes the set of edges that connect tasks to the data they produce; and $E_{DT} \subseteq V_D \times V_T$ denotes the edges that connect tasks to the data they consume. Figures 3.1 and 3.2 show examples of the task graphs for the edge detection and convolutional neural network applications.

Given a task graph G, we can obtain the following auxiliary definitions that will be used later.

$$
\begin{array}{llcll}
(a) & PI & = & \{d \in V_D \,|\forall t \in V_T,\ e_{td} \notin E_{TD}\} & \text{: primary inputs} \\
(b) & PO & = & \{d \in V_D \,|\forall t \in V_T,\ e_{dt} \notin E_{DT}\} & \text{: primary outputs} \\
\forall t \in V_T & & & & \\
(c) & ID(t) & = & \{d \in V_D \mid e_{dt} \in E_{DT}\} & \text{: input data to task } t, \\
(d) & OD(t) & = & \{d \in V_D \mid e_{td} \in E_{TD}\} & \text{: output data of task } t,
\end{array}
$$

$\forall t \in V_T$

$$(e) \qquad O(t) = \{t' \in V_T \mid \exists\, d \in V_D \;:\; d \in OD(t) \wedge d \in ID(t')\} \quad : \text{tasks dependent on task } t$$

$\forall d \in V_D$

$$(f) \qquad P(d) = \begin{cases} t : e_{dt} \in E_{DT} & \text{if } d \notin PI \\ \epsilon & \text{if } d \in PI \end{cases} \quad : \text{task that produces data } d$$

$$(g) \qquad U(d) = \{t \in V_T \mid e_{dt} \in E_{DT}\} \quad : \text{tasks that use data } d$$

Architecture Model

The architecture consists of a CPU and a GPU connected by a bus. The only architectural parameter that affects the mapping is the total memory size M of the GPU. We assume that the CPU memory is large enough to hold all application data.

Task graph annotations

We annotate all data items V_D in the task graph G with the size of their footprint in memory. This is denoted by the function $s : V_D \rightarrow \Re^+$, which assigns a non-negative size $s(d)$ to each data item d.

3.2.2 Optimization Problem

Variables

There are two sets of primary variables that we use in the optimization problem. The first of these encodes the ordering of tasks on the GPU. Given the application model $G = (V_T, V_D)$, data sizes s and GPU memory limit M, we define a function $L : V_T \rightarrow \{1, 2, \ldots |V_T|\}$ which maps every task $t \in V_T$ to the position (or *level*) of the task in the global task schedule order. The assumption here is that all tasks execute on the GPU. The function L is an onto function: each task is assigned to a single level.

The second set of primary variables encodes the data that are present on the GPU after each task has been executed. Let $GPU(d, l)$ be a binary variable that is 1 if data d is present in GPU memory at level l in the task schedule order and takes the value 0 otherwise. $GPU(d, l)$ presents a snapshot of the data resident in GPU memory immediately after the task at level l executes.

The L and GPU variables together define the solution space of the problem. However, for the ease of representing the constraints of the problem, we also define the following auxiliary variables that define the data transfers to and from GPU memory. We define the binary variable $IN(d, l)$ to be 1 if data item d is transferred *into* the GPU just *before* the task at level l executes. We also define $OUT(d, l)$ to be 1 if data item d is transferred out of the GPU just *after* the task at level l executes.

Constraints

For the task schedule L to be valid, it must obey the precedence constraints in the task graph.

$$\forall t_1 \in V_T, \forall t_2 \in O(t_1) \text{ (dependence constraints)},$$
$$(a)\ L(t_2) \geq L(t_1) + 1$$

Given a valid schedule level L, we can compute the liveness of the data items. We follow the same definition of liveness as the definition in the register allocation problem in compiler optimization [Goodwin and Wilken, 1996]. A data item $d \in V_D$ is said to be live at a particular level l if the following constraint is satisfied:

$$l_{min}(d) \leq l \leq l_{max}(d)$$

where

$$l_{min}(d) = \begin{cases} L(P(d)) & \text{if } d \notin PI \\ 0 & \text{otherwise} \end{cases}$$

$$l_{max}(d) = \begin{cases} \max\{L(u) : u \in U(d)\} & \text{if } d \notin PO \\ |V_T| + 1 & \text{otherwise} \end{cases}$$

This says that a data element d becomes live at the level l_{min} where either the task that produces d ($P(d)$) executes, or if d is a primary input to the application, at the beginning of the application. Similarly, data d stops being live at a level l_{max} where either all tasks that use it finish executing, or if d is a primary output of the application, the end of the application.

The set of tasks that are live at a particular level need to be present either in GPU or CPU memory. Since most data excepting the primary inputs and outputs are both produced and consumed on the GPU, it is preferable to keep as much of the data as possible in GPU memory. The following constraints need to be satisfied on the $GPU(d, l)$ variables that define the set of data present in GPU

memory at any given level l:

$$\forall d \in V_D \text{ (initial state constraints)},$$
$$(b)\ GPU(d, 0) = 0$$
$$\forall d \in V_D \text{ (final state constraints)},$$
$$(c)\ GPU(d, |V_T| + 1) = 0$$
$$\forall t \in V_T,\ \forall d \in\ (ID(t) \cup OD(t))\ \text{ (input/output constraints)},$$
$$(d)\ GPU(d, L(t)) = 1$$

Constraint (b) says that no data is present in GPU memory at the beginning of the application. Constraint (c) imposes a similar restriction at the end of the application. This models the common usage model of CPU-GPU systems, where input data is produced on the CPU, sent to the GPU for processing and the results transferred back from the GPU. Constraint (d) ensures that the inputs and outputs of a task that has just finished executing should be present in GPU memory.

The memory size of the GPU limits the data that can be present in the GPU at any point of time. The following constraint enforces that all live data present on the GPU at any level fits into GPU memory. Here $ID(t)$ and $OD(t)$ are the input and output data of task t, $s(d)$ is the size of data s, and M is the GPU memory size.

$$\forall l \in \{1, 2, \ldots . |V_T|\} \text{ (memory size constraints)},$$
$$(e)\ \textstyle\sum_{d:l_{min}(d) \leq l \leq l_{max}(d)} (s(d) \cdot GPU(d, l)) \leq M$$

The data that is live at any level but is not present on the GPU at a particular level must have been transferred out of the GPU before that level. Such data must then be transferred back to the GPU when they are required as inputs to an executing task. The following constraints determine the values of the $OUT(d, l)$ and $IN(d, l)$ variables that define whether data element d is transferred out or into the GPU at level l:

$$\forall d \in V_D,\ \forall l \in \{1, 2, \ldots . |V_T|\},\ (l - 1) \geq l_{min}(d) \text{ (input data transfer constraints)},$$
$$(f)\ \neg\, GPU(d, l - 1) \wedge GPU(d, l) \Rightarrow IN(d, l)$$
$$\forall d \notin PI,\ \forall l \in \{1, 2, \ldots . |V_T|\},\ (l + 1) \leq l_{max}(d) \text{ (output data transfer constraints)},$$
$$(g)\ GPU(d, l) \wedge \neg\, GPU(d, l + 1) \wedge (\textstyle\bigwedge_{l' < l} \neg\, OUT(d, l')) \Rightarrow OUT(d, l)$$

Constraint (f) specifies that if data d is not present at level $l - 1$ but is present at level l, and if d was live at level $l - 1$, then it must have been brought in to the GPU at level l. The condition that data d must be live at level $l - 1$ prevents data transfers in case data d is produced at level l. Note that it is possible for a data element to be repeatedly brought back to GPU memory from the CPU;

each such transfer would be represented by a different $IN(d, l)$ variable being set to 1. Constraint (g) specifies that if a data item is present in the GPU at level l and not at level $l + 1$ and it is live at $l + 1$, then it must be transferred out of the GPU. However, we assume that the CPU memory is large enough to retain all the data that it ever receives from the GPU for the entirety of execution of the application. Hence, data that has already been transferred once from the GPU to the CPU need never be transferred out again. Such data can be overwritten in GPU memory, and can be recalled from CPU memory at a later point if necessary. This is encoded in our constraint by ensuring that we do not set $OUT(d, l) = 1$ if $OUT(d, l') = 1$ at any previous level l'. For a similar reason, it is also never necessary to transfer the primary inputs of the application to the CPU, even if it will be required again at a later point of time. Therefore we do not impose the output data transfer constraint for primary inputs.

The primary inputs to the application are initially present in CPU memory and the primary outputs need to be transferred to CPU memory. Constraints (b), (d) and (f) ensures that the primary inputs are transferred into the GPU when they are first used. Constraints (c), (d) and (f) together ensure that the primary outputs of the application are transferred to the CPU at or before level $|V_T|$. This is because the primary outputs are present in the GPU at the level they are produced (from (c)), but are not present in the GPU at level $|V_T + 1|$ (from (d)). Hence constraint (g) is invoked at some level between the point of production and the end of the program, thereby causing the transfer out of the GPU.

Optimization Objective

The optimization objective is to minimize the sum of the sizes of all data transferred to and from the GPU.

$$\min \sum_{d \in V_D} \sum_{l \in \{1,2,...|V_T|\}} s(d) \cdot (IN(d, l) + OUT(d, l))$$

subject to constraints (a) through (g) described previously.

Properties of the optimal solution

The above optimization problem is expressed in terms of the primary variables of L, the task ordering and GPU, the data present on the GPU after each task executes. It turns out that we can reduce the solution space of the problem by instead expressing it in terms of the L and a subset of the $IN(d, l)$ variables, where the IN variables represent the transfers into the GPU just before the

execution of each task. We prove below that only those input transfers $IN(d,l)$ where the data d is an input to the task at level l need to be considered as part of the solution space. Further, we can prove that we do not need to include the OUT and GPU variables in the solution space if we include these IN variables. This can significantly reduce the solution space and thereby increase the efficiency of solution techniques.

We begin by defining $Use(d,l)$ for a data item d and a schedule level l to be 1 if data d is an input at level l. This will be the case if any of the tasks that use d execute at level l.

$$Use(d,l) = \begin{cases} 1 & \text{if } l \in \{L(u) : u \in U(d)\} \\ 0 & \text{otherwise} \end{cases}$$

We now note the following properties:

Theorem 3.2.1 *There exists an optimal solution to the data transfer minimization problem such that* $IN(d,l) = 0$ *whenever* $Use(d,l) = 0$.

Proof Proof is by contradiction. Assume that there is a optimal solution where $IN(d,\lambda) = 1$ when $Use(d,\lambda) = 0$ at some level λ. Consider all $\lambda' > \lambda$ in increasing order until either we reach a point, say at level k, where $Use(d,k) = 1$ or we reach $\lambda' = |V_T|$, the end of the application. In the former case, scheduling $IN(d,k) = 1$ instead of $IN(d,\lambda) = 1$ will have the same transfer cost, but using λ instead of k unnecessarily increases the region where data item d is present in GPU memory. This can lead to unnecessary transfers of other data from GPU memory. Replacing λ by k can thus never increase the overall optimization cost, and hence $IN(d,\lambda)$ is not required for the optimal solution. In the latter case when $\lambda' = |V_T|$, the data item d is never used in the program at point λ or beyond, and hence it is unnecessary to ever transfer it to the GPU. A solution to the data transfer minimization problem containing $IN(d,\lambda)$ cannot be optimal. ∎

Theorem 3.2.2 *There exists an optimal solution to the data transfer minimization problem such that*

$$OUT(d,l) = \begin{cases} 1 & \text{if } (d \notin PI) \wedge (l = L(P(d))) \wedge \left((\bigvee_{l=1}^{|V_T|} IN(d,l)) \vee (d \in PO) \right) \\ 0 & \text{otherwise} \end{cases}$$

Proof We consider two cases: $d \in PI$ and $d \notin PI$. A data element $d \in PI$ never needs to be transferred to the CPU, since it is already present in CPU memory at the beginning of the application. This is covered by the second case in the theorem.

Consider $d \notin PI$. If $d \in PO$, the d always needs to be transferred out of the CPU at some point. Also, for all other data elements, if $IN(d,l) = 1$ for any l, then data d must be present in the CPU at level l; hence there must exist some point before l at which $OUT(d,l) = 1$. In the following, we show that it is sufficient to transfer d only when $l = L(P(d))$, i.e. as soon as it is produced (case 1). Further, we show that if d is neither a primary output nor has it been transferred in from the CPU at some point, then the data need not be transferred to the CPU at all (case 2).

We show this in two parts: (a) we first show that $OUT(d,l) = 0$ whenever $l \neq L(P(d))$, and (b) we show that $OUT(d,l) = 0$ whenever $\bigvee_{l=1}^{|V_T|} IN(d,l) = 0$ and $d \notin PO$.

Proof of claim (a) is by contradiction. Assume that there is a optimal solution with $d \notin PI$ and where $OUT(d,l) = 1$ when $l \neq L(P(d))$. Consider all $l' < l$ in decreasing order until either we reach a point, say at level k, where $k = L(P(d))$ or we reach $l' = 0$, the beginning of the application. In the former case, scheduling $OUT(d,k) = 1$ instead of $OUT(d,l) = 1$ will have the same transfer cost, but using l instead of k can unnecessarily increase the region where data item d is present in GPU memory. This can lead to unnecessary transfers of other data from GPU memory. Replacing l by k can thus never increase the overall optimization cost, and hence l is not required for the optimal solution. In the latter case when $l' = 0$, the data item d has not been produced by the tasks at level l or before, and hence cannot be transferred out. Such a schedule with $OUT(d,l) = 1$ is not valid.

We now only need to prove claim (b) for $l = L(P(d))$. We note that since d is not a primary input or output, it is only used internally by tasks in V_T. Under these circumstances, the transfer $OUT(d,l)$ is only necessary if it is transferred into the GPU at some future point. However, $\bigvee_{l=1}^{|V_T|} IN(d,l) = 0$ implies that $IN(d,l) = 0 \forall l \in \{1, 2, \ldots |V_T|\}$. This means that d is never transferred in. Hence we can remove the transfer $OUT(d,l)$, if present, without making the resulting schedule invalid, thereby strictly reducing the cost. This invalidates our assumption that our initial schedule was optimal. ∎

Theorem 3.2.3 *There exists an optimal solution to the data transfer minimization problem such that:*

$$GPU(d,l) = \begin{cases} 1 & \text{if } l \in \{L(u) : u \in U(d)\} \cup L(P(d)) \\ 1 & \text{if } l_{min}(d) \leq l \leq l_{max}(d) \wedge \neg IN(d, Next\_Use(d,l)) \\ 0 & \text{otherwise} \end{cases}$$

where

$$Next\_Use(d,l) = \min\{l' > l : l' \in \{L(u) : u \in U(d)\}\}$$

Proof The first line follows from the input/output constraint (d). We need to prove the other two lines.

To prove that $GPU(d,l) = 1$ if d is live at level l and $\neg IN(d, Next\_Use(d,l))$, we note that from the contra-positive of constraint (f), $\neg IN(d,l) \Rightarrow GPU(d,l-1) \vee \neg GPU(d,l)$. But by definition, for $k = Next\_Use(d,l)$, d is used at level k, and hence $GPU(d,k) = 1$ (from constraint (d)). Hence, from the contra-positive of (f) at level k, we are left with $GPU(d,k-1) = 1$. If $l = k-1$, then we are done. Otherwise, we note that d is not transferred into the GPU between level l and k (by theorem 1). Hence, we continue applying the contra-positive of (f) at levels $k-1, k-2, \ldots l+1$ to obtain $GPU(d,l) = 1$.

To prove that in all other cases, $GPU(d,l) = 0$, we first note that if d is not live, then clearly $GPU(d,l) = 0$. If d is live and $IN(d, Next\_Use(d,l)) = 1$, then we prove that $GPU(d,l) = 0$ by contradiction. Assume that $GPU(d,l) = 1$. Then we can construct a new schedule by changing $GPU(d,l) = 0$. Since data d is not used between l and $Next\_Use(d,l)$ (by definition), the new schedule is still valid at levels between l and $Next\_Use(d,l)$. For all levels at or above $Next\_Use(d,l)$, the schedule and the set of data in GPU memory remains the same as in the original schedule. Data d is still present in the GPU at level $Next\_Use(d,l)$ because $IN(d, Next\_Use(d,l)) = 1$. Since the schedule is valid and does not lead to any additional transfers, it is still optimal. ∎

Theorem 3.2.4 *The schedule levels L and the transfers into the GPU $IN(d,l)$ (subject to Theorem 3.2.1) are sufficient to fully describe the optimization problem.*

Proof The only other variables in constraints (a) - (f) involve the $OUT(d,l)$ and $GPU(d,l)$ variables. We have shown in Theorems 3.2.2 and 3.2.3 that these can be defined in terms of the L and $IN(d,l)$ variables. ∎

The above theorems restrict the effective solution space of the problem to possible schedule levels L and possible $IN(d,l)$ values where d is an input to the task at level l. We use these theorems to simplify the optimization techniques that we present in the next section.

3.3 Techniques for Static Optimization

In this section, we discuss techniques to solve the data transfer minimization problem.

3.3.1 Previous Work

The problem of minimizing data transfers in the GPU context has only been recently studied [Sundaram *et al.*, 2009]. The authors propose a heuristic to perform task ordering and data transfer scheduling to minimize transfers.

The problem of minimizing data transfers has, however, been studied in other contexts. Register allocation is a well-known problem routinely solved by optimizing compilers. The problem of *local register allocation* is to allocate variables within a basic block to hardware registers in such a way as to minimize spillage to memory. The local register allocation problem was proved to be NP-hard in [Farach and Liberatore, 1998]. Many exact and approximate techniques have been developed for register allocation [Goodwin and Wilken, 1996] [Farach and Liberatore, 1998] [Liberatore *et al.*, 1999]. However, local register allocation generally works with a fixed sequence of instructions. This problem is then similar to our problem under a fixed task ordering. The same problem of minimizing data transfers is also studied in the context of minimizing transfers to scratchpad memory [Verma *et al.*, 2004]. This problem also, however, works with a fixed task ordering.

The problem of finding optimal task orderings has been studied in the context of instruction reordering and task scheduling (Chapter 2). Most of these approaches optimize for application execution time and not data transfers. A related problem to our work is the problem of finding an optimal instruction ordering for minimizing register usage. This problem has been proven to be NP-hard in [Govindarajan *et al.*, 2003]. The authors propose an exact method and a heuristic to solve this problem. However, the problem they consider is still simpler than our problem as it merely seeks to find the minimum number of registers required to avoid spills. The problem of finding the minimum amount of spillage under a fixed register size is harder from a theoretical computational complexity perspective than this problem. For instance, under a fixed instruction ordering, the former problem can be solved in polynomial time while the latter is NP-hard [Farach and Liberatore, 1998].

The data transfer minimization problem as outlined in the previous sections is NP-hard. The problem of local register allocation can be reduced to this problem by fixing a particular task ordering for our problem. Since the local register allocation problem is known to be NP-hard, so is our problem. Further, the decision version of this problem is in the complexity class NP. The decision version of the problem asks whether it is possible to find a valid task and data transfer schedule such that the total data transfer size is smaller than a given constant. Given a certificate of a task and data transfer, it is a polynomial time algorithm to (1) check that the certificate represents a valid

schedule (check that it satisfies all the polynomial number of constraints of the problem) and (2) if the schedule is valid, then compute the total transfer size and check that it is less than the given constant. Therefore the decision version of this problem is NP-complete.

Although the problem is NP-complete, it is not necessarily the case that we will not be able to solve this problem. We now study a range of exact and approximate techniques in a bid to solve this problem.

3.3.2 Exact MILP formulation

We can express the problem as a Mixed Integer Linear Programming (MILP) problem and use commercial MILP solvers such as the CPLEX engine from ILOG [ILOG Inc.,] to solve it. From our experiments on other scheduling problems (Chapter 2), we expect that a single pass MILP solution (where all problem constraints are presented at the start of execution) will not be able to handle large scale problems. Nevertheless, an MILP-based approach can usually solve small-scale problems to optimality. For instances where the problem is not optimally solved, an MILP based approach can deliver consistent improvements to a bound for the optimal solution over time. Additionally, such an approach has the advantage of leveraging a commercial tool and can reap the benefits of regular solver improvements over time.

The variables and constraints for the MILP formulation are obtained from the problem description in Section 3.2.2. We have the option of adding additional constraints corresponding to the results of Theorems 3.2.1- 3.2.3 to reduce the effective solution space. It is not always the case that adding these constraints speeds up the time taken by the solver. There is usually a trade-off between the size of the solution space encoded in the problem and the actual number of variables and constraints presented to the solver. In this section, we attempt to encode all the information in Section 3.2.2 into our formulation.

We start by considering the result of Theorem 3.2.4. According to this theorem, the primary decision variables for the data transfer optimization problem are those used to encode the task schedule and the input data transfers. We define the binary variables:

$\forall t \in V_T, \forall l \in \{1, 2 \ldots |V_T|\},$

$$x_{SL}(t,l) : \begin{cases} 1 \text{ if schedule level of task } t = l \\ 0 \text{ o.w.} \end{cases}$$

$\forall d \in V_D, \forall l \in \{1, 2 \ldots |V_T|\},$

$$x_{IN}(d,l) : \begin{cases} 1 \text{ if data } d \text{ is transferred into the GPU just before the task at level } l \\ 0 \text{ o.w.} \end{cases}$$

We impose the following constraints on the x_{SL} variables:

$$\begin{aligned} &\forall l \in \{1, 2 \ldots |V_T|\}, && (D1) \quad \textstyle\sum_{t=1}^{|V_T|} x_{SL}(t,l) = 1 \\ &\forall t \in V_T, && (D2) \quad \textstyle\sum_{l=1}^{|V_T|} x_{SL}(t,l) = 1 \\ &\forall t_1 \in V_T,\ \forall t_2 \in O(t_1),\ \forall l_1, l_2 \in \{1, 2, \ldots |V_T\}, l_2 > l_1, \\ & && (D3) \quad x_{SL}(t_1,l_2) + x_{SL}(t_2,l_1) \leq 1 \end{aligned}$$

Constraints (D1) - (D3) are responsible for obtaining a valid task ordering. (D1) forces each level to be occupied by any one task and (D2) assigns each task to some level. Constraint (D3) restricts the valid task orderings to respect the precedence constraints in the task graph.

We next consider Theorems 3.2.1 to 3.2.3 in sequence. As per Theorem 3.2.1, we can restrict the $x_{IN}(d,l)$ variables to be set to 1 only if data d is an input to the task executing at level l. This constraint is imposed as:

$$\forall d \in V_D, \forall l \in \{1, 2 \ldots |V_T|\}, \quad (IN1) \quad x_{IN}(d,l) \leq \sum_{u \in U(d)} x_{SL}(u,l)$$

By Theorem 3.2.2, there is only one possible level at which a given data d can be transferred to the CPU. This is known to be the level at which the task producing the data executes. A consequence of this theorem is that the optimization problem does not need to solve for the exact level at which each data item is transferred out of the GPU. We can instead merely optimize for whether a given data item d is to be transferred out at any point of the program or not. If it is transferred out, the task

schedule encoded in the values of the x_{SL} variables will allow us to easily obtain the exact position of the transfer. We define the binary variable:

$$\forall d \in V_D, \quad x_{OUT}(d) : \begin{cases} 1 \text{ if data } d \text{ is transferred out of the GPU at any level} \\ 0 \text{ o.w.} \end{cases}$$

From Theorem 3.2.2, we can obtain the value of x_{OUT} as:

$$\begin{aligned} &\forall d \in PO, && (O1) \quad x_{OUT}(d) = 1 \\ &\forall d \in V_D, d \notin PI, d \notin PO, \forall l \in \{1, 2, \ldots |V_T|\}, && (O2) \quad x_{OUT}(d) \geq x_{IN}(d, l) \end{aligned}$$

Condition (O1) asserts that all primary outputs are always transferred out. Condition (O2) says that if a task is not a primary input or output, then it is transferred only if it has been transferred in at some level.

It now remains to encode the presence of the data in GPU memory using Theorem 3.2.3. In order to do this, we must first define the following binary variables:

$\forall d \in V_D, \forall l \in \{1, 2, \ldots |V_T|\},$

$$\left.\begin{array}{lcl} x_P(d,l) & : & 1 \text{ if data } d \text{ has been produced at or before level } l \\ x_C(d,l) & : & 1 \text{ if data } d \text{ has been used by all tasks before level } l \\ x_L(d,l) & : & 1 \text{ if data } d \text{ is live at level } l \\ x_{USE}(d,l,l') & : & 1 \text{ if data } d \text{ is used after level } l \text{ and at or before level } l' \\ x_{NU}(d,l,l') & : & 1 \text{ if the next use of data } d \text{ after level } l \text{ is at level } l' \\ x_{IN\_NU}(d,l) & : & 1 \text{ if } x_{IN}(d,l') = 1 \text{ where } l' \text{ is the next use of data } d \text{ after level } l \end{array}\right\} \text{and 0 o.w.}$$

The following constraints are responsible for defining the liveness variables x_P, x_C and x_L.

$$\begin{aligned} \forall d \in PI, \quad & (L1) \quad x_P(d,0) = 1 \\ \forall d \notin PI, \quad & (L1') \quad x_P(d,0) = 0 \\ \forall d \in PO, \quad & (L2) \quad x_C(d, |V_T|) = 0 \\ \forall d \notin PO, \quad & (L2') \quad x_C(d, |V_T|) = 1 \\ \forall d \notin PI, \forall l \in \{1, 2 \ldots |V_T|\}, \quad & (L3) \quad x_P(d,l) \geq x_{SL}(P(d), l) \\ \forall d \in V_D, \forall l \in \{1, 2 \ldots |V_T|\}, \quad & (L4) \quad x_P(d,l) \geq x_P(d, l-1) \\ \forall d \notin PO, \forall l \in \{1, 2 \ldots |V_T|\}, & \\ \forall u \in U(d) \quad & (L5) \quad x_C(d,l) + x_{SL}(u,l) \leq 1 \end{aligned}$$

$$\forall d \in V_D, \forall l \in \{1, 2 \ldots |V_T|\}, \quad (L6) \quad x_C(d,l) \leq x_C(d,l+1)$$
$$\forall d \in PO, \forall l \in \{1, 2 \ldots |V_T|\} \quad (L7) \quad x_L(d,l) \leq x_P(d,l),\ x_L(d,l) + x_c(d,l) \leq 1$$
$$\forall d \in PO, \forall l \in \{1, 2 \ldots |V_T|\} \quad (L8) \quad x_L(d,l) \geq x_P(d,l) - x_C(d,l)$$

Of these, constraints (L1) and (L1') set the initial conditions for the x_P variable. Constraints (L2) and (L2') do the same for the x_C variable which indicates when data has been fully consumed. Constraints (L3) sets the $x_P(d,l)$ variable to 1 if d is produced at level l. Constraint (L4) propagates the values of all $x_P(d,l)$ variables set to 1 to successor levels. Constraint (L5) sets $x_C(d,l)$ to 0 if data d is used at level l, thereby implying that not all tasks that use d finish before level l. Constraint (L6) propagates this information to previous levels. Finally, constraints (L7) and (L8) define data d to be live if it $x_P(d,l) = 1$ and $x_C(d,l) = 0$.

The x_{USE}, x_{NU} and x_{IN_NU} variables are set by the following constraints:

$$\forall d \in V_D, \forall l \in \{1, 2 \ldots |V_T|\}, \quad (U1) \quad x_{USE}(d,l,l) = 0$$
$$\forall d \in V_D, \forall l \in \{1, 2 \ldots |V_T|\}, \forall l' > l,$$
$$(U2) \quad x_{USE}(d,l,l') \geq x_{USE}(d,l,l'-1)$$
$$(U3) \quad x_{USE}(d,l,l') \geq \sum_{u \in U(d)} x_{SL}(u,l')$$
$$\forall d \in V_D, \forall l \in \{1, 2 \ldots |V_T|\}, \forall l' > l,$$
$$(NU1) \quad x_{NU}(d,l,l') \leq x_{USE}(d,l,l')$$
$$(NU2) \quad x_{NU}(d,l,l') + x_{USE}(d,l,l'-1) \leq 1$$
$$(NU3) \quad x_{NU}(d,l,l') \geq x_{USE}(d,l,l') - x_{USE}(d,l,l'-1)$$
$$\forall d \in V_D, \forall l \in \{1, 2 \ldots |V_T|\}, \forall l' > l,$$
$$(INNU1) \quad x_{IN\_NU}(d,l) \geq x_{IN}(d,l') + x_{NU}(d,l,l') - 1$$
$$\forall d \in V_D, \forall l \in \{1, 2 \ldots |V_T|\}, \quad (INNU2) \quad x_{IN\_NU}(d,l) \leq \sum_{l'>l} z(d,l')$$
$$\text{where } z(d,l') \leq x_{IN}(d,l'),\ \ z(d,l') \leq x_{NU}(d,l,l')$$

Constraints (U1)-(U3) express the constraint: $x_{USE}(d,l,l') = \bigvee_{u \in U(d), l \leq k \leq l'} x_{SL}(u,k)$. Constraints (NU1) and (NU2) define $x_{NU}(d,l,l') = x_{USE}(d,l,l') \wedge \neg x_{USE}(d,l,l'-1)$. Constraints (INN1) and (INN2) define the quantity $x_{IN\_NU}(d,l) = x_{IN}(d, Next\_Use(d,l'))$ as used in the definition of Theorem 3.2.3.

Finally, we define the binary variable:

$$\forall d \in V_D, \forall l \in \{1, 2 \ldots |V_T|\}, \quad x_{GPU}(d,l): \text{ 1 iff data } d \text{ is present in the GPU at level } l$$

and the corresponding constraints:

$$\begin{array}{lll}
\forall d \in V_D, \forall l \in \{1, 2 \ldots |V_T|\} & (G1) & x_{GPU}(d,l) \geq x_{USE}(d,l) \\
\forall d \in V_D, \forall l \in \{1, 2 \ldots |V_T|\} & (G2) & x_{GPU}(d,l) \geq x_{SL}(P(d),l) \\
\forall d \in V_D, \forall l \in \{1, 2 \ldots |V_T|\} & (G3) & x_{GPU}(d,l) \geq x_L(d,l) - x_{IN_NU}(d,l) \\
\forall d \in V_D, \forall l \in \{1, 2 \ldots |V_T|\} & (G4) & x_{GPU}(d,l) \leq x_{USE}(d,l) + x_{SL}(P(d),l) + y(d,l) \\
 & & \text{where } y(d,l) \leq x_L(d,l), y(d,l) \leq \big(1 - x_{IN_NU}(d,l)\big) \\
 & & \\
\forall l \in \{1, 2 \ldots |V_T|\} & (M) & \sum_{d=1}^{|V_D|} s(d) \cdot x_{GPU}(d,l) \leq M
\end{array}$$

Constraints (G1) - (G4) define $x_{GPU}(d,l)$ as in Theorem 3.2.3. Constraint (M) imposes the memory size constraint of the GPU.

Finally, we must ensure that the primary inputs and outputs are transferred into and out of the GPU. Constraint (O1) takes care of transferring the primary outputs back to the CPU. The primary inputs, on the other hand, have to be handled separately:

$$\forall d \in PI, \quad (IN2) \quad x_{IN\_NU}(d,0) = 1$$

This constraint ensures that the primary input is transferred into the CPU at its first use.

The objective of the optimization is:

$$\min \sum_{d \in V_D} \sum_{l \in \{1,2,\ldots|V_T|\}} s(d) \cdot (x_{IN}(d,l) + x_{OUT}(d,l))$$

where $s(d)$ represents the size of the data element d.

The problem encoding involves $O(|V_D||V_T|^2)$ variables and constraints. The scales of the problems we look at range from $|V_T|, |V_D| \sim 10$'s to 1000's. This can lead to problems in the range of millions of variables and constraints. Commercial integer programming solvers like CPLEX can solve the small examples with $|V_T|, |V_D|$ around 50-100 to completion, but do not perform well on the large examples. In many cases, the solver does not find the solution to even the LP relaxation of the problem. Thus there is a need to reduce the size of the problem in terms of the variables and constraints. This motivates the need to break up this problem into simpler sub-problems that are solved more easily.

3.3.3 Decomposition-based Approaches

A common way to simplify complex optimization problems is to decompose the problem into simpler sub-problems. A natural decomposition of our problem would be to separate out the optimization of the task schedule from the optimization of data transfers. However, these two sub-problems are inter-related: we need to know the task schedule in order to find the optimal data transfers. Moreover, we cannot evaluate the optimality of a task schedule without solving the data transfer optimization problem. A brute-force exact approach to resolving this inter-dependence is to iterate over all possible task orders and solve the data transfer optimization sub-problem exactly for each possible task order. This is, of course, not a feasible approach since the number of task orderings grows exponentially in the number of tasks. Alternatively, we can sample the solution space of possible task orders using heuristics or simulated annealing, and only solve the data transfer optimization on those task orders. Such a technique would be more feasible in terms of runtime, but we lose the guarantee of optimality of the resulting schedule. We follow the latter approach in Section 3.3.5. Before we study techniques to sample the solution space of task orders, we focus on the sub-problem of finding the optimal data transfer schedule given a task order. This is the subject of Section 3.3.4.

3.3.4 Data transfer scheduling given a task order

Given a task ordering x_{SL}, we can simplify the optimization problem in Section 3.3.2. As a first step, the value of the schedule levels x_{SL} is given, and hence constraints (D1)-(D3) that deal with obtaining a task order can be omitted.

Under a fixed task ordering, many of the variables in the exact formulation become constants that can be pre-computed. In Theorem 3.2.1, we can determine the values of $Use(d, l)$ for each (d, l) pair. We can use these pre-computed $Use(d, l)$ values to eliminate all $x_{IN}(d, l)$ variables where $Use(d, l) = 0$. We do this by simply replacing these variables with the constant 0 wherever they appear in the MILP formulation. In Theorem 3.2.3, both the liveness of variables (represented as the x_L variables in Section 3.3.2 as well as the position of the next use of a data item $Next\_Use(d, l)$ can be computed as per their definitions in Section 3.2.2 and Theorem 3.2.3 respectively. This eliminates the need for the liveness constraints (L1)-(L8) and the set of constraints (U1)-(U3), (NU1)-(NU3) and (INNU1)-(INNU2) that compute the $Next\_Use$ variable.

Finally, we can use Theorem 3 to eliminate the need for expressing the x_{GPU} variables. We know that each input and output data d for the task executing at each level l must be present in the

GPU at that level (Constraint (G1)). Instead of imposing this as a constraint, we replace the value of such x_{GPU} variables by the constant 1 wherever they occur in the formulation of the problem. We also know that for (d, l) pairs where data d is not live at level l, $x_{GPU}(d, l) = 0$ (Constraint (G2)). Again, we do not create x_{GPU} variables for such (d, l) pairs and replace them by the constant 0. For all other (d, l) pairs, Theorem 3.2.3 indicates that $x_{GPU}(d, l) = \neg x_{IN}(d, Next\_Use(d, l))$. As $Next\_Use(d, l)$ is a constant, we can replace occurrences of x_{GPU} variables with the expression $\big(1 - x_{IN}(d, Next\_Use(d, l))\big)$.

After performing all the simplifications outlined above, the resulting data transfer problem has the sets of variables:

$$\forall (d, l) \in V_D \times \{1, 2 \ldots |V_T|\} \ : \ l \in \{L(u) : u \in U(d)\},$$

$x_{IN}(d, l)$: 1 iff data d is transferred into the GPU just before the task at level l executes

$\forall d \in V_D$

$x_{OUT}(d)$: 1 iff data d is transferred out of the GPU immediately after the task at level $L(P(d))$ executes

The constraints for the problem are:

$$\forall l \in \{1, 2 \ldots |V_T|\}$$

$$(M') \quad \sum_{\substack{d \in ID(t) \cup OD(t) \\ : x_{SL}(t,l)}} s(d) + \sum_{\substack{d \notin ID(t) \cup OD(t) \\ : x_{SL}(t,l) \\ \wedge\, l_{min}(d) \leq l \leq l_{max}(d)}} s(d) \cdot \Big(1 - x_{IN}(d, Next\_Use(d, l))\Big) \leq M$$

$$\forall d \in PO$$

$$(O1) \quad x_{OUT}(d) = 1$$

$$\forall d \in V_D, d \notin PO, \forall l \in \{1, 2, \ldots |V_T| \mid l \in \{L(u) : u \in U(d)\}\ \}$$

$$(O2) \quad x_{OUT}(d) \geq x_{IN}(d, l)$$

$$\forall d \in PI$$

$$(I1) \quad x_{IN}(d, Next\_Use(d, 0)) = 1$$

Here constraint (M') is the equivalent of the memory size constraint (M) $\sum_{d=1}^{|V_D|} s(d) \cdot x_{GPU}(d, l) \leq M$ of Section 3.3.2, which states that the sum of data sizes of all data on the GPU must be smaller than the GPU memory size. We obtain (M') from (M) by replacing the x_{GPU} values according to the discussion previously in this section. Constraints (O1) and (O2) represent the results of

Theorem 3.2.2, and are carried forward from Section 3.3.2. Constraint (I1) is the equivalent of constraint (IN2) from Section 3.3.2. Together, constraints (I1) and (O1) ensure that primary inputs are transferred into the GPU and primary outputs are transferred out of the GPU.

The objective, as before, is to minimize the total transfers:

$$\min \sum_{d \in V_D} s(d) \cdot \left(x_{OUT}(d) + \sum_{l \in \{ L(u) : u \in U(d) \} } x_{IN}(d,l) \right)$$

This optimization problem only has variables of the order of the number of edges E in the task graph G, since each input edge to a task represents one point where a data d is used, and hence to one $x_{IN}(d,l)$ variable. The number of constraints is of the order of $O(E + V_T)$, which is linear in the input graph. This optimization problem is considerably smaller than the problem of simultaneously optimizing both the task schedule and data transfers. Even for large problem sizes with a few thousands of tasks and data items, the problem of finding the optimum data transfer schedule given a task schedule can be solved to completion in a fraction of a second using commercial solvers such as CPLEX.

Apart from computational feasibility, the formulation also aids us in our understanding of the problem. Constraint (M') clearly shows the impact of data being transferred into the GPU on the data that needs to be kept in the GPU. In particular, if $x_{IN}(d,l)$ is always 0, the left hand side of the inequality blows up to the sum of sizes of all data that is live at each level. We can thus obtain the intuitive result that unless the GPU memory size is big enough to hold all live data at each level, it is necessary for some input transfers to occur. Constraints (O1) and (I1) explicitly show that data transfers of the primary inputs and outputs are compulsorily required irrespective of the task schedule. This establishes a lower bound on the result of the optimization.

3.3.5 Finding a good task ordering

In order to minimize data transfers, tasks must be ordered in such a way as to minimize the time for which data items are live. By definition, a data item is live from the time it is produced until the time that all tasks that use this data complete.

The problem of finding the optimal task order for our problem is NP-complete. We can reduce the problem of optimizing instruction reordering to minimize register usage [Govindarajan *et al.*, 2003] to this problem. Indeed, the decision version of the instruction reordering problem asks if it possible to order a set of instructions with dependence constraints so as to use no more than K registers, for some fixed K. We can solve this problem by mapping instructions to tasks and

registers to memory locations, and ask if it is possible to order the tasks so as to produce code with 0 spills, given a memory bound of K. This is the decision version of our problem with all data assumed to have unit sizes.

Heuristic Approaches

Sundaram et al. propose a heuristic to order the tasks according to a *depth-first* traversal of the directed acyclic graph G [Sundaram *et al.*, 2009]. Their reasoning is that a depth-first ordering of a graph ensures that data along only one path of the graph from the inputs to the outputs needs to be kept in memory. The procedure for finding such a task order is given in Algorithm 3.1. The input to the algorithm is the task graph G. The output is a valid task order encoded by the levels L at which tasks execute. The algorithm uses a visited array that keeps track of whether a task has already been scheduled or not. This is initialized to 0 for all tasks (line 1). The algorithm works by scheduling tasks to execute at increasing levels starting with level 0. The current level is stored in a $level$ variable (line 2). The algorithm starts by scheduling tasks that use the primary inputs (line 3-4). For each such task, the algorithm calls a recursive procedure that is responsible for finding the schedule levels of all tasks in the sub-tree of G rooted at that task (line 5). Finally, the levels L updated by the recursive procedure is returned (line 6).

The recursive procedure is outlined in Algorithm 3.2. This algorithm takes in the graph, the task u whose sub-tree needs to be scheduled, the $visited$ array, the current $level$ and the currently updated set of schedule levels L. It updates the $visited$, $level$ and L variables. The procedure first checks if the node u has already been scheduled, and if so, exits the procedure (line 1). If u is not yet scheduled, the procedure then checks to see if all predecessors of u have been scheduled; if not it returns (line 2). This ensures that the schedule order returned obeys the task precedence edges in the graph: no task is scheduled at a level unless all predecessors are scheduled at earlier levels. If all predecessors of u have already been scheduled, then $L(u)$ is set to the current value of $level$, which is then incremented (line 3). The task is then stored as having been scheduled (line 4). The procedure is then recursively called for each data item that is produced by u (line 5). For each such data d, all tasks that use d are attempted to be scheduled successively (line 6-7).

Algorithm 3.1 embodies a heuristic task ordering algorithm for the data transfer minimization problem that may not yield optimal solutions. As an example, the task ordering obtained by the algorithm for the edge detection application of Figure 3.1 is shown in Figure 3.5. The optimal data transfers given this task schedule is also shown in the figure. For the example data sizes and

Algorithm 3.1 DFS(G) $\rightarrow$ L

```
1  Set visited(t) = 0 ∀t ∈ V_T
2  Set level = 0
3  forall (d ∈ PI)
       // run depth first search from the tasks that use d
4      forall u ∈ U(d)
5          DFSRECURSIVE(G, u, visited, level, L)
6  return L
```

Algorithm 3.2 DFSRECURSIVE(G, u, $visited$, $level$, L)

```
1  if visited(u) = 1 return
2  if (visited(v) = 1 ∀v : u ∈ O(v))
3      L(u) = level, level++
4      visited(u) = 1
5      forall w ∈ O(u)
6          DFSRECURSIVE(G, w, visited, level, L)
7  return
```

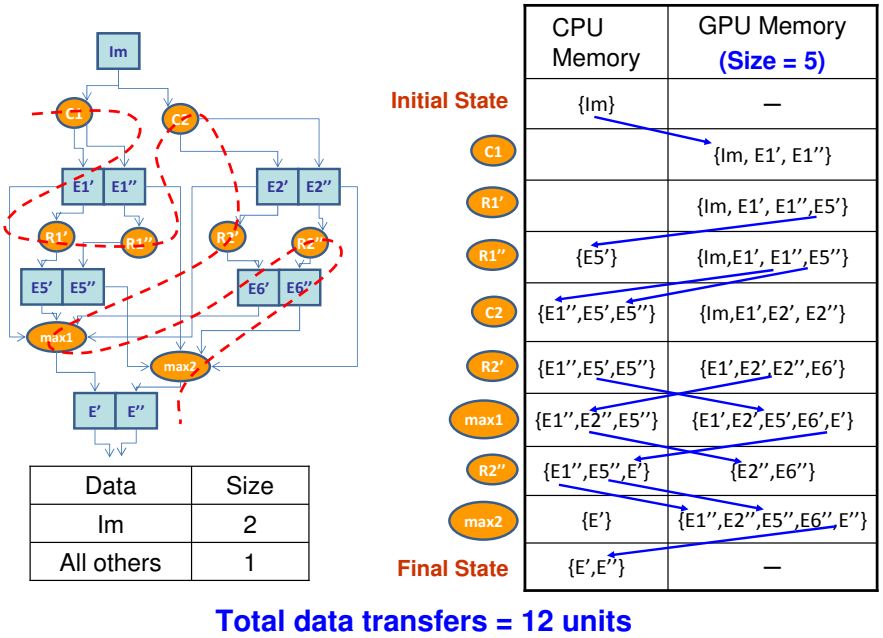


Figure 3.5: Task order and resulting data transfers obtained from the DFS heuristic for the task graph in Figure 3.1.

memory size shown, we can see that this task schedule requires 12 units of data to be transferred. As a comparison, the task order in schedule Figure 3.4 only requires 8 units of data transfer. The task schedule in Figure 3.4 is the optimal schedule, but is more irregular than the DFS heuristic

schedule.

It is important to realize that the presence of task precedence constraints prevents the task order returned by Algorithm 3.1 from purely being a depth-first order. For instance, in Figure 3.5, tasks R_1' and max_1 use data item E_1'; hence DFSRECURSIVE is called successively on tasks R_1' and max_1. However, max_1 can start execution only after R_2' completes, and is thus not available for execution at that point. The algorithm instead proceeds (in a depth-first fashion) to next execute task R_1'', and then C_2, R_2' and so on. Thus the ordering obtained by the algorithm is $C_! \rightarrow R_1' \rightarrow R_1'' \rightarrow C_2 \rightarrow R_2' \rightarrow max_1 \rightarrow \ldots$. It turns out that this modified depth-first order does not help reduce the number of live variables in this example. It is better instead to avoid executing R_1'' in between R_1' and C_2, as in the schedule shown in Figure 3.4.

The problem of finding the optimal task ordering is NP-complete. As such, it is not surprising to find that a heuristic solution that finds a single valid task order according to some greedy criterion is sub-optimal. An alternative approach is to explore the space of valid task orders according to a simulated annealing algorithm. Such an algorithm was also used to explore the space of valid allocations of tasks to processors and task schedules in the multiprocessing scheduling problem of Chapter 2.

Simulated Annealing

Simulated Annealing is a generic approach to solve many combinatorial optimization problems. Simulated Annealing works by making moves (transitions) over a state space and evaluating a cost metric at each state. The basic structure of a simulated annealing algorithm was described in Algorithm 2.3 in Chapter 2. As we noted in Section 2.3.2, a particular implementation of simulated annealing tunes the selection of the COST, MOVE, PROB and TEMP functions, and the initial and final temperature parameters t_0 and t_∞.

In Section 2.3.2, we described the use of simulated annealing and the choice of the parameters and functions in the context of the task allocation and scheduling for multiprocessors. The solution space of our current problem (all valid task orderings) is similar to the solution space of the multiprocessor scheduling problem, with the exception that all tasks execute on the single GPU in the system and hence task allocation to processors is not a concern. The similarity in the state space of the two problems means that a similar choice of functions and parameters as Section 2.3.2 should be expected to yield good results for the current problem as well.

In the current work, we reuse the PROB, TEMP function and the initial and final temperatures

t_0 and t_∞ from Section 2.3.2. The COST and MOVE functions are the only ones we change. These are chosen as follows:

- COST function: The COST function $Cost(s)$ specifies the value of the optimization function at state s of the simulated annealing. In the current context, a state encodes a particular global ordering of tasks, subject to task precedence constraints (Section 3.2.2). The COST function for such a state is the value of the optimum data transfer from the CPU to the GPU given this task order. We obtain this by solving the Mixed Integer Linear Program as described in Section 3.3.4. The time taken for the COST evaluation is a fraction of a second.

- MOVE function: The MOVE function defines the transitions in the state space of simulated annealing. Our MOVE function is based on a random move of one task from its initial position in the global order to a new position. Let s be the current state defined by the task schedule levels L. The MOVE function then picks a random task $v \in V_Y$ and selects a new position l' for the task. It then inserts task v into position l' by shifting all tasks currently scheduled at or after position l'. In other words, the task schedule levels are updated to L' such that $L'(v) = l'$, and $L'(w) = L(w) + 1, \forall w : L(w) \geq l'$. To ensure that this defines a valid task order, we need to check that there are no predecessors of v at a level greater than l', and that there are no successors of v at a level less than l'. If the new state does not define a valid task schedule, it is discarded and a new random task schedule is picked again.

3.4 Results

In this section, we present the results of our experiments to evaluate different techniques of solving the task and data transfer scheduling problem to minimize makespan. We compare the "single-pass" Mixed Integer Linear Programming (MILP) approach with two decomposition based approaches. Both decomposition based approaches break up the problem into a task ordering problem followed by a data transfer scheduling problem given the task order. The first decomposition approach constructs a heuristic task order based on the depth-first heuristic described in [Sundaram *et al.*, 2009](Section 3.3.5), and then solves a MILP problem for obtaining the schedule of data transfers given the task order. We denote this approach as (DFS/MILP). The second approach performs a simulated annealing (SA) search over the space of task orders and solves a MILP problem for each task order that it evaluates. This is denoted as the (SA/MILP) approach. The single-pass MILP was given a timeout of 20 minutes, and the best solution obtained until that time was taken

as the solution of the problem. All experiments were run on a 2.4 GHz machine with 8 GB RAM running Linux.

Name	# Tasks	# Data Items	Input image resolution	Size of total intermediate data (words)	Size of PI + PO (words)
Edge detection	8	13	1000×1000	5841800	1968768
			5000×5000	149201800	49840768
			10000×10000	598401800	199680768
CNN1	740	1134	4500×3800	3,729,863,824	315,144,108
			6400×4800	6,709,288,924	566,690,708
			8000×6000	10,491,402,124	885,961,908
CNN2	1600	2434	4500×3800	3,478,703,106	278,714,152
			6400×4800	6,261,866,429	501,282,002
			8000×6000	9,795,926,781	783,798,202

Table 3.1: Benchmark characteristics for evaluating different scheduling techniques for minimizing data transfers.

The benchmarks used in our comparison were the edge detection and convolutional neural network applications. The task graphs for both applications were obtained from the authors of [Sundaram *et al.*, 2009]. For each of these applications, we experimented with input data of different sizes. For the edge detection application, the input data corresponds to images of high resolution. In our experiments, we used input images of sizes of 1000×1000, 5000×5000 and 10000×10000. For the Convolutional Neural Network (CNN) application, we used two different networks with differing numbers of layers, tasks and data items. The first of these has 7 layers with 740 tasks and 1134 data items, while the second CNN has 11 layers with 1600 tasks and 2434 data items. For each CNN, we use data images of sizes 4500×3800, 6400×4800 and 8000×6000. The key characteristics of the benchmarks are summarized in Table 3.1. For each benchmark, we show the size of the input image, the size of all data in the benchmark (expressed in units of 32-bit words) and the size of the primary inputs and outputs of the benchmark (in words). We use three GPUs of varying memory sizes: the NVIDIA 8800 GT with 512 MB memory, the NVIDIA 8800 GTX with 768 MB memory and the NVIDIA Tesla C870 platform with 1.5 GB memory.

Table 3.2 shows the results of the three scheduling approaches on the data transfers on each of our benchmarks. The first two columns state the benchmark and the input image resolution used. The third column is the number of floating point data transfers required for the primary inputs and outputs. This provides a lower bound to the number of transfers required for the benchmark. The rest of the columns show the results in terms of the number of floating point data transfers obtained

Benchmark	Input image resolution	Lower bound (words)	8800 GT (512 MB memory)		
			MILP	DFS/ MILP	SA/ MILP
Edge detection	1000×1000	1,968,768	**1,968,768**	1,968,768	1,968,768
	5000×5000	49,840,768	**49,840,768**	49,840,768	49,840,768
	10000×10000	199,680,768	∞	∞	∞
CNN1	4500×3800	315,144,108	-	596,249,820	451,148,460
	6400×4800	566,690,708	-	2,463,812,372	1,469,753,036
	8000×6000	885,961,908	-	∞	∞
CNN2	4500×3800	278,714,152	-	559,819,864	314,774,228
	6400×4800	501,282,002	-	2,398,403,666	1,062,256,223
	8000×6000	783,798,202	-	∞	∞

8800 GTX (768 MB memory)			Tesla C870 (1.5 GB memory)		
MILP	DFS/ MILP	SA/ MILP	MILP	DFS/ MILP	SA/ MILP
1,968,768	1,968,768	1,968,768	**1,968,768**	1,968,768	1,968,768
49,840,768	49,840,768	49,840,768	**49,840,768**	49,840,768	49,840,768
∞	∞	∞	**199,680,768**	299,361,024	199,680,768
-	315,144,108	315,144,108	-	315,144,108	315,144,108
-	1,607,411,540	941,433,176	-	566,690,708	566,690,708
-	4,067,974,476	2,705,681,316	-	1,508,786,916	1,244,702,988
-	278,714,152	278,714,152	-	278,714,152	278,714,152
-	1,542,002,834	547,160,618	-	501,282,002	501,282,002
-	3,965,810,770	2,275,595,289	-	1,406,623,210	926,847,914

Table 3.2: Data transfer optimization results for the single-pass MILP approach and two decomposition approaches: a depth-first search heuristic combined with MILP (DFS/MILP) and a simulated annealing search combined with MILP (SA/MILP) on different benchmarks on three GPU platforms.

(in units of 32-bit words) from the three scheduling approaches on three different NVIDIA GPUs with different on-board memory sizes. The MILP technique is stopped after twenty minutes, and the best result obtained until that time is taken. We indicate the entries where the MILP approach finds an optimal result within 20 minutes in bold. The remaining approaches are run to completion (all instances completed under twenty minutes). For data inputs that are very large, it is possible for the inputs and outputs of a single task to not fit into GPU memory. For such cases, the application cannot be mapped onto the GPU. We represent such cases with a "∞".

We can see that the single-pass MILP approach can give us optimal results for the edge detection application which has only 8 tasks and 13 data items. However, the MILP approach does not scale to larger task graphs, and in many cases does not even find the solution to the LP relaxation

of the problem within the time bound of 20 minutes. We represent such cases where no solution to the problem is found by ”-” in the table. From Table 3.2, we can see that the single-pass approach cannot solve any of the medium scale or large scale CNN cases.

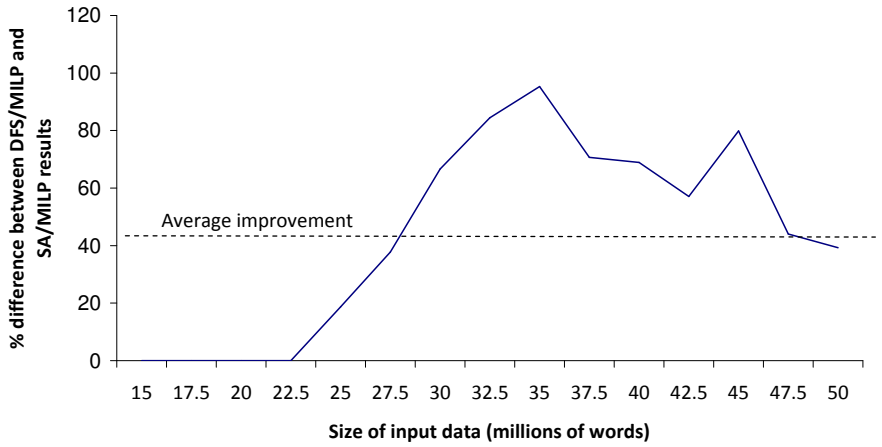


Figure 3.6: Percentage difference between the data transfers obtained from the DFS/MILP and SA/MILP approaches for the CNN1 benchmark in Table 3.2 optimized for the 8800 GTX platform.

For the larger test cases, both the DFS/MILP and the SA/MILP approaches can be used to minimize data transfers. The DFS/MILP heuristic approach is successful at identifying cases where no additional data transfers are required on top of the lower bound of the primary inputs and outputs. This generally happens for benchmarks that use small to medium resolution input data images. For the larger resolution images, additional data transfers become necessary as the intermediate data does not fully fit into GPU memory. In this case, we find that the heuristic approach gives significantly worse results than the simulated annealing approach. Figure 3.6 shows the percentage difference between the memory transfers obtained by the DFS/MILP and SA/MILP approaches for the CNN1 benchmark with the 8800 GTX GPU. We can see that the SA/MILP approach routinely finds solutions that are 40-60% better than the DFS/MILP approach. The average improvement for the data shown in the graph is 44.2%.

For the edge detection example, both the single-pass MILP and the SA/MILP approaches are able to obtain the optimal solution. However, we do not know how close the SA/MILP results are to the optimal solutions for the larger CNN examples. In order to judge the optimality of the SA/MILP approach for the large examples, we compared it to a variant of the SA/MILP method that was modified to increase the number of transitions at each annealing temperature by a factor of 100. This latter technique, henceforth called “SA/MILP (long)” was allowed to run overnight. We

expect the result of SA/MILP (long) to be fairly close to the optimal solution. Hence this forms a good benchmark solution for our optimization methods. Table 3.3 shows the results of comparing the SA/MILP approach to the SA/MILP (long) approach for the CNN benchmarks. The table shows the percentage differences between the amount of data transfers computed by the SA/MILP and the SA/MILP (long) algorithms. For data inputs that are very large, it is possible for the inputs and outputs of a single task to not fit into GPU memory. For such cases, the application cannot be mapped onto the GPU. We represent such cases with a "∞". The entries marked 0 indicate the cases where the result of SA/MILP is not improved even when it is run overnight. All such entries correspond to the cases where the SA/MILP results are optimal (this can be seen by comparing the results to the lower bound in Table 3.2). For the cases where the result of SA/MILP is not optimal, we generally see an improvement when the technique is run overnight. This improvement can be in the range of 5-15%. If we can assume that the SA/MILP (long) solution is close to optimal, then the result of the SA/MILP algorithm is up to 15% away from optimal. Another way of looking at this result is that we can gain up to 15% in the size of data transferred by allowing the SA/MILP approach to run longer. The choice of whether a longer runtime (of the order of a few hours) is acceptable or not depends on the context in which this optimization problem is solved. If the optimization problem is solved inside of a design space exploration over the space of different GPUs, then we may not be able to devote a few hours to each run. In that case, it would be acceptable to lose 10-15% in solution optimality. However, if we only wish to solve the optimization problem for a single GPU platform, then we can afford to let the optimization run overnight.

Benchmark	Input image resolution	% difference from SA/MILP and SA/MILP (long)		
		8800 GT	8800 GTX	Tesla C870
CNN1	4500×3800	3.4	0	0
	6400×4800	13.6	0.8	0
	8000×6000	∞	10.8	8.3
CNN2	4500×3800	6.3	0	0
	6400×4800	15.8	12.3	0
	8000×6000	∞	14.9	10.5

Table 3.3: Percentage difference between the data transfer results of the SA/MILP approach and a variant SA/MILP (long) that is allowed to run overnight on different benchmarks on three GPU platforms. This is used as a measure of the optimality of the SA/MILP approach.

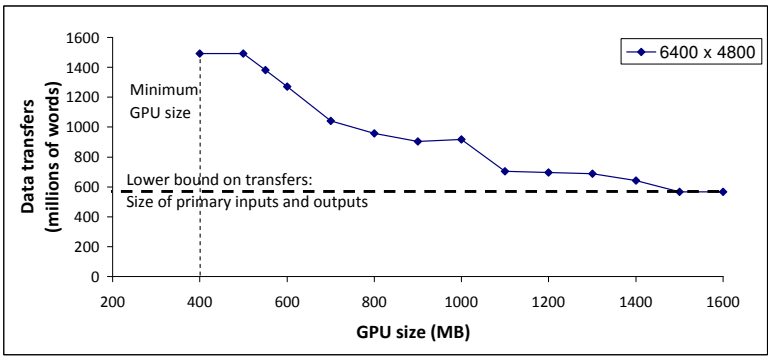
The other dimension to look at when deciding which algorithm to use is the run time taken to obtain these solutions. We only compare the two decomposition methods under this metric, since these are the methods consistently applicable to problems of large size. Table 3.4 reports the time

Benchmark	Input image resolution	8800 GT		8800 GTX		Tesla C870	
		DFS/ MILP	SA/ MILP	DFS/ MILP	SA/ MILP	DFS/ MILP	SA/ MILP
Edge detection	1000×1000	0	0	0	0	0	1
	5000×5000	0	0	0	1	0	0
	10000×10000	0	0	0	0	0	0
CNN1	4500×3800	1	346	1	472	1	443
	6400×4800	1	238	1	306	1	427
	8000×6000	0	0	1	523	1	365
CNN2	4500×3800	1	1023	1	834	1	974
	6400×4800	1	736	1	1129	1	1012
	8000×6000	0	0	1	966	1	825

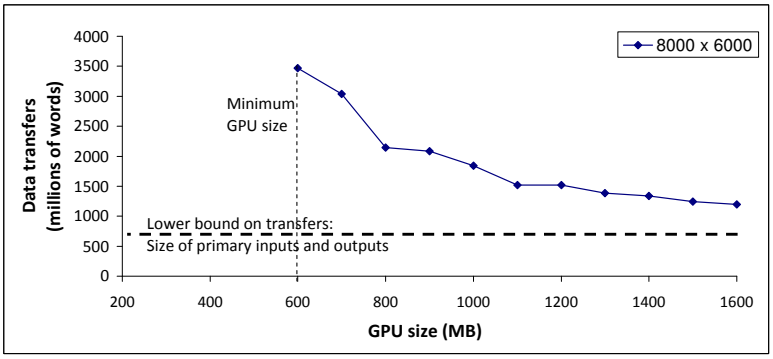
Table 3.4: Run times (in seconds) of the DFS/MILP and the SA/MILP decomposition approaches corresponding to the entries of Table 3.2 on different benchmarks on three GPU platforms.

taken by the two methods for each of our benchmarks of Table 3.2. As expected, the heuristic approach only considers a single task order, and as such always completes in at most a second. The SA/MILP approach, on the other hand, explores many task orders. However, we find that for all our cases, it does finish within twenty minutes, which is a reasonable time-frame to optimize an application.

The solutions to the data transfer problem can be used in the choice of GPUs to be used for a particular application. Figure 3.7 shows the amount of data transfers (along with the lower bound) for the first CNN application with 6400×4800 and 8000×6000 resolution images versus the memory size of the GPU. We can see that the graphs for both the resolution images follow the same trends: until the GPU size reaches a certain size, the input/output data for a single task does not fit into GPU memory. Thus we cannot run the application on such GPUs. This is marked as the minimum GPU size on the graphs. As we increase the size of the GPU beyond that point, the size of the required data transfers decreases gradually until it reaches a lower bound. The lower bound corresponds to the transfer of primary inputs and outputs, which must be performed irrespective of the GPU memory size. This establishes a maximum GPU memory size that can lead to reductions in data transfers. The minimum and maximum GPU sizes so obtained help decide the range of GPUs to be used for input data of different sizes. In this example, it is possible to use the 8800 GT card with 512 MB RAM for the 6400×4800 application, but not for the larger 8000×6000 resolution images. For the larger resolution images, it is best to use the 1.5 GB NVIDIA Tesla card. If cost or power considerations rule out using the Tesla card, then it is also possible to use the 8800 GTX card at the cost of about a 2.2X increase in the number of memory transfers.



(a)



(b)

Figure 3.7: Data transfers obtained from the SA/MILP approach for the CNN1 benchmark for input image resolutions of (a) 6400×4800 and (b) 8000×6000 optimized for GPUs of different sizes.

3.5 Choice of Optimization Method

In this chapter, we studied the optimization problem of resource allocation of data to GPU and CPU memory and scheduling the data transfers between them in order to minimize data transfers between the CPU and GPU. We outlined three techniques for solving the data transfer optimization problem. The single-pass MILP solution does not scale beyond 50 tasks and data items, and hence is not applicable to medium and large scale problems. The two decomposition approaches SA/MILP and DFS/MILP are viable techniques. In this work, we found that the sub-problem of deciding a schedule for data transfers given a task order can be solved to optimality very quickly using a commercial MILP solver. The remaining problem is to pick a task order. For this problem, we found that the SA/MILP approach that performs a simulated annealing search over the set of task orders is superior (by an average of 44.2%) to the DFS/MILP approach that uses a heuristic

depth-first task order. We also found that the SA/MILP approach can optimize data transfers for large task graphs containing thousands of tasks and data items in under twenty minutes. Hence the simulated annealing algorithm gives us the best mix of quality of results versus optimization time for optimizing data transfers.

As an additional benefit, we note that the parameters for tuning the simulated annealing algorithm as described in this chapter are very similar to the techniques described in Chapter 2 for the task allocation and scheduling problem. This enables a quick development of the simulated annealing algorithm for this problem, and points to the extensibility of using simulated annealing as an optimization engine.

Chapter 4

Statistical Models and Analysis for Task Scheduling

In Chapter 2, we discussed the task allocation and scheduling problem using compile-time knowledge of the characteristics of the concurrent application. Optimization techniques used for compile-time scheduling are useful when we have complete knowledge (at compile time) of (a) the computation tasks, (b) task dependencies, and (c) task execution times. The underlying assumption in the task graph model we use in Chapter 2 is that execution times and communication delays of each task are fixed real-valued constants. This model is popularly referred to as the *delay model* in scheduling literature [Papadimitriou and Ullman, 1987] [Boeres and Rebello, 2003]. The delay model has been used in many multiprocessor scheduling problems [Hu, 1961] [Coffman, 1976] [Papadimitriou and Ullman, 1987] [Veltman *et al.*, 1990] [El-Rewini *et al.*, 1995] [Teich and Thiele, 1996] [Dick *et al.*, 2003].

A fundamental limitation of the delay model is that it does not capture the variability in task execution times and dependencies. Task execution times can vary significantly across different runs in many real-world applications, due to (a) input data dependent executions of loops or conditionals that are present within the code, (b) the effect of contention and arbitration for shared resources, (c) effects of cache hits or misses on memory access times in a multi-level memory hierarchy, and (d) the increasing effect of process variations that leads to variable clock frequencies among different cores of a multi-core architecture. In some applications (such as H.264 video decoding), we may not even have complete knowledge of the dependencies in the application during compile-time. Such variations in individual tasks lead to variations in the resulting end-to-end execution time (or

makespan, defined in Section 2.2.2) of the overall application. In particular, the makespan of the application can no longer be accurately represented as a single number. We instead need to capture the variability in the makespan by the *statistical distribution* of finish times of the application.

It is possible to obtain better estimates of task execution times at run-time with knowledge of the inputs to the application. However, variability due to the architecture involving cache misses and bus arbitration effects cannot be easily predicted even at run-time. Shih et al. perform a workload characterization of the H.264 video decoder and conclude that "unpredictable branch behavior appears to be the main performance bottleneck in the (H.264) application" [Shih *et al.*, 2004]. Holliman et al. develop a performance analysis of MPEG-2 and H.264 decoders on the Pentium architecture and independently corroborate that "branches for variable-length decoding are often essentially random and performance-limiting" [Holliman *et al.*, 2003]. Building compile-time models for systems also has other benefits for design space exploration of different architectures. In many cases, it may be too costly or even impossible to build an executable model of each system to be evaluated. In such cases, an accurate compile-time model of the system can help in early pruning of the design space [Gries, 2004].

In the presence of variations in execution times, compile-time models for applications that require strict execution time guarantees rely on capturing either the worst-case behavior of tasks. Such applications are often called *hard real-time* applications. For applications that do not require hard real-time guarantees, the dominant execution scenarios may instead be captured in the models. These approaches approximate the distribution of application performance by the worst-case or common-case scenarios. In contrast, a large class of recent applications fall under the category of *soft real-time* applications. Soft real-time applications do not define application performance in terms of the absolute worst case execution scenario, but instead use a statistical metric to define a scenario that covers a high fixed percentage of all executions. Examples of such applications include video encoders/decoders in the multimedia domain, packet forwarders and network address translation in the networking domain and many other streaming applications. Since the performance of the application does not reflect the worst-case execution scenario, it is possible under certain conditions for the application to not complete within its expected execution time. Soft real-time applications must therefore be tolerant to incomplete processing of up to a fixed percentage of data (frames in video applications, packets in networking applications). For instance, it may be acceptable for a H.264 video decoder to only decode 95% of the input frames and drop the remaining 5%. In this scenario, the execution time of the decoder is best measured as the 95^{th} *percentile* of the distribution of frame execution times rather than the worst or average case. Such an estimate of

performance has the advantage of being robust to the worst case execution time, which may not be truly representative of real application performance.

In order to obtain the performance distribution of the application at compile-time, it is essential to model the variations in task execution times. Thus the static delay model is insufficient for our purposes. In this chapter, we explore the types of variations present in applications in the networking and multimedia domains. We will consider two specific benchmarks - IPv4 packet forwarding and H.264 video decoding. We propose *statistical models* to capture these variations and perform *statistical analysis* on these models to produce the application makespan distributions. We compare the statistical analysis of these two applications to worst-case and average case analysis. We will then revisit the task allocation and scheduling problem in the context of statistical models in Chapters 5 and 6.

4.1 Variability in application execution times

In this section, we shall study some of the causes of variability in application execution times. The variability in application makespan is a consequence of the variability of the execution times of each task in the application. Since the execution time of a task depends on both the code inside the task as well as the platform that executes the code, a natural classification of the variability is into application and architecture related causes.

Many application tasks exhibit variable execution times due to data-dependent executions on different inputs. The extent of the variability depends on how much control flow there is within the task in the form of data-dependent branches and loop iterations. Many applications in the networking [Gupta, 2000] [Yu *et al.*, 2005] and media [Hughes *et al.*, 2001] [Shih *et al.*, 2004] domains exhibit such variations. Hughes et al. demonstrated a 33% variability in frame execution times for a range of video encoding/decoding applications on a single processor core [Hughes *et al.*, 2001]. Gupta et al. studied the impact of the variable number of memory accesses required to lookup IPv4 packets with different source IP addresses [Gupta, 2000]. In addition, the dependencies and interactions between tasks may also be variable in the presence of varying inputs. Video decoders such as H.264 [Chong *et al.*, 2007] are good examples. We will explore this source of variability in Section 4.2.1.

The presence of shared resources in the architecture results in variations in memory access and communication times. The effect of caches on task execution times has been well studied [Agarwal *et al.*, 1988] [Cucchiara *et al.*, 1999] [Slingerland and Smith, 2001]. Caches can cause variations due

to data-dependent cache hits and misses. Variations can also occur due to bus arbitration effects. Kim et al. demonstrated that contention among a set of 16 processors sharing the same bus can result in a 5-fold increase in communication times on bus loads common to multimedia applications [Kim *et al.*, 2005]. Finally, the increasing extent of process variations as we move to smaller process technology nodes has led to variability in execution time even between the cores of a multiprocessor system.

We now consider two examples in the networking and multimedia domains: the IPv4 packet forwarding application when mapped to a soft multiprocessor system on an FPGA, and the H.264 video decoding algorithm mapped onto a commercial many-core processor.

4.1.1 IPv4 packet forwarding on a soft multiprocessor system

We discussed the IPv4 packet forwarding application in brief in Chapter 2. In that chapter, we also defined a soft multiprocessor system on an FPGA. We construct soft multiprocessor systems out of the MicroBlaze soft processor core, the On-Chip Peripheral (OPB) buses to connect these processors to on-chip BlockRAM memory and Fast Simplex Links (FSLs) to provide point-to-point links between the processors [Ravindran *et al.*, 2005] [Jin *et al.*, 2005].

The basic IP forwarding application consists of the following steps (Section 2.1): (a) receive the packet, (b) verify the packet checksum and Time-to-Live (TTL) fields, (c) lookup the IP next hop and forwarding port using longest prefix matching on a route table, (d) update the checksum and TTL fields, and (e) send out the packet. The complete data plane of the IPv4 application must also deal with packet payload transfer. However, the header processing component, in particular the next hop lookup, is the most compute intensive data plane operation. The packet payload is buffered on chip and transferred to the egress port after the header is processed. Since all data processing occurs only on the header, the resulting forwarding throughput only depends on the header processing component.

Figure 4.1 replicates the task graph described in Chapter 2 for IPv4 packet forwarding. The tasks in the top branch perform the memory lookups in the IPv4 packet forwarding application. The other tasks are receive, which receives the packet, tasks for verifying the IP checksum and time-to-live fields, tasks to update the checksum and time-to-live fields, and the transmit task that sends the packet to the next hop.

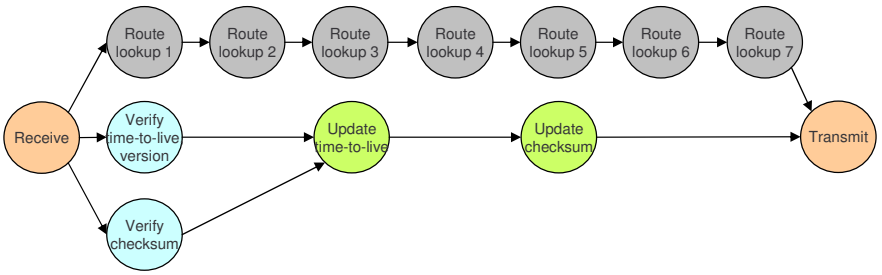
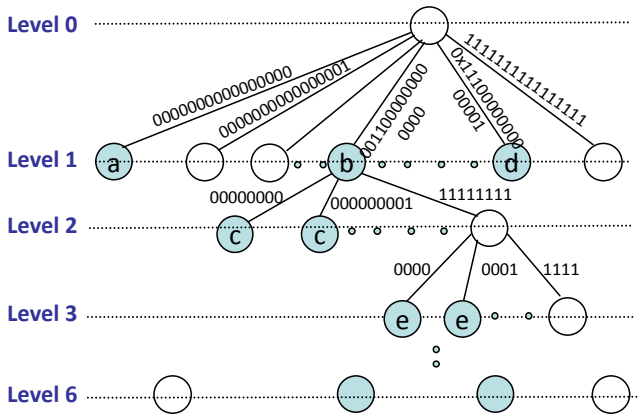


Figure 4.1: Parallel tasks and dependencies in the IPv4 header processing application.

Variations due to application characteristics

The next hop lookup is the most intensive operation in the application, and this is also where most of the variability in the application comes from. The lookup involves searching a route table for the longest prefix that matches the packet's destination address. A natural way to express prefixes is a tree-based data structure (called a *trie*) that uses the bits of the prefix to direct branching. There have been many variations to the basic trie scheme that attempt to trade-off the memory requirements of the trie table and the number of memory accesses required for lookup [Ruiz-Sánchez *et al.*, 2001]. Figure 4.2 shows an example of a fixed-stride multi-bit trie table and the route table that it encodes. The stride is the number of bits inspected at each step of the prefix match algorithm. The stride order we use in our implementation (shown in Figure 4.2) is (12 8 4 3 3 2): the first-level stride of the trie inspects the first 12 bits of the destination IP, the second level inspects the next 8 bits and so on, leading to a maximum of 6 memory accesses for a 32-bit address lookup. These lookups take us to a matching node for the destination address in the trie; there is one last memory access required for obtaining the egress port, making for a total maximum of 7 lookups. In figure 4.1, lookup stages 1 to 6 depict the 6 possible lookups to get to a leaf of the trie and lookup stage 7 represents the final compulsory port lookup.

Due to the nature of the longest prefix match, the lookup algorithm can often reach a matching node while performing fewer than 7 lookups. The number of bits required to be looked up depends on the IP address and the distribution of prefixes in the route trie table. Figure 4.3 shows the prefix length distribution of the trie table of a typical backbone router (Telstra router) taken from [Ruiz-Sánchez *et al.*, 2001]. We note from the figure that most prefixes are between 16 and 24 bits in length, with less than 4% being more than 24 bits, and an even smaller fraction (less than 2%) being more than 28 bits. Kim et al. independently corroborate that over 99% of another router



Prefix	IP Address match	Output port
a	0000000000000000*	1
b	0011000000000000*	5
c	0011000000000000--0000000*	2
d	1110000000000001*	6
e	0011000000000000--11111111000*	3

Figure 4.2: Example of a multi-bit trie with stride order (12 8 4 3 3 2) and the corresponding route table.

(MAE-EAST) have prefixes smaller than 24 bits [Kim and Han, 2005]. Similar characteristics also exist for smaller campus routers [Oh and Lee, 2003]. From this prefix distribution, we can obtain the probability distribution for the number of memory accesses required for our multi-bit trie. Any prefix that is shorter than 12 bits will be at level 1 of the trie and requires only one memory access. With a prefix length greater than 12 bits, more than one access will be required. The probability distribution of the number of memory accesses is shown in Figure 4.4. This probability distribution is a histogram of the number of memory lookups required for each router trie table entry assuming that each entry is accessed equally often. The assumption that each trie table entry is accessed equally often is commonly used in networking applications as exact data traffic characteristics are not readily available for privacy reasons [Gupta, 2000] [Kim and Han, 2005].

# memory accesses	Probability	Execution Time of Lookup Stages						
		Stage 1	Stage 2	Stage 3	Stage 4	Stage 5	Stage 6	Stage 7
2	0.0816	20	0	0	0	0	0	20
3	0.1131	20	20	0	0	0	0	20
4	0.7744	20	20	20	0	0	0	20
5	0.0124	20	20	20	20	0	0	20
6	0.0155	20	20	20	20	20	0	20
7	0.0030	20	20	20	20	20	20	20

Table 4.1: Joint probability distribution table for tasks Lookup 1-6 of Fig 4.1 assuming that each memory lookup takes 20 cycles.

The execution times of each lookup task can be represented as a binary distribution depending on whether the particular lookup stage is active or not. Figure 4.4 naturally yields the execution

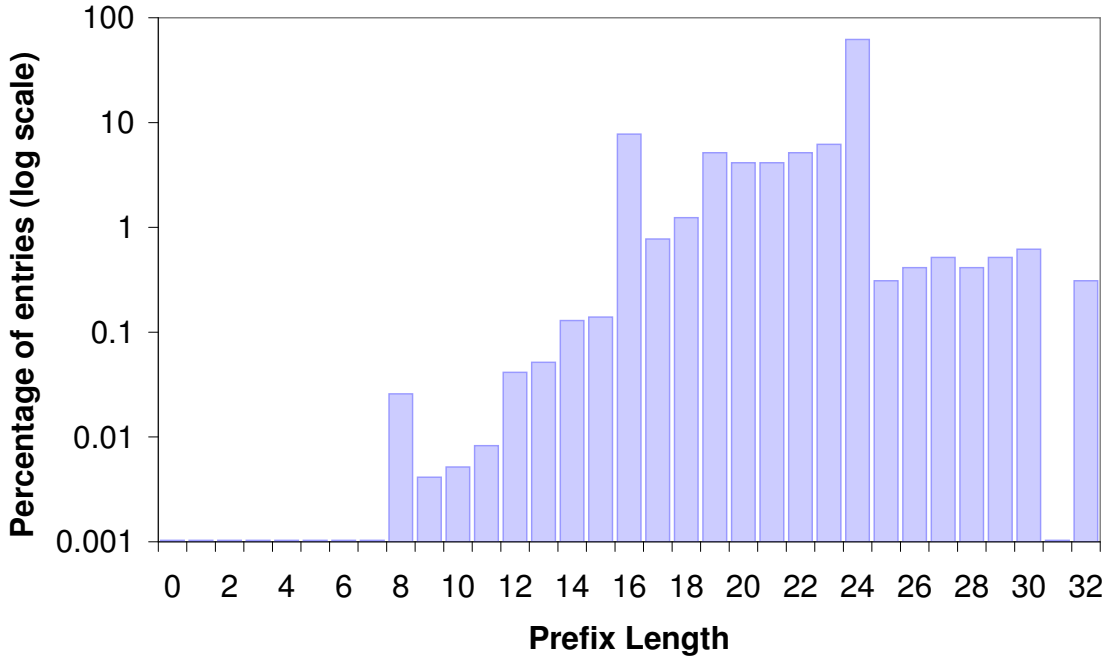


Figure 4.3: Prefix length distribution of a typical backbone router (Telstra Router, Dec 2000) from [Ruiz-Sánchez *et al.*, 2001].

time distribution of each of the 7 forwarding tasks in the task graph of Figure 4.1. From Chapter 2, each of the memory lookup tasks has a 20-cycle latency. The first lookup always occurs and hence the Lookup 1 task has to perform a 20-cycle latency memory lookup with 100% probability. Lookup 2 is required whenever at least 3 memory lookups are required; this occurs 91.8% of the time from Figure 4.4. Lookup 2 is inactive (with a execution time of 0) the remaining 8.2% of the time. The remaining lookup stages will have corresponding distributions that can be computed from Figure 4.4. However, it is important to realize that the execution times of these tasks cannot vary independently; in statistical terms, the execution times of the tasks are said to be *correlated* random variables. This is because whenever a lookup is done at a particular level of the trie table, lookups at previous levels must also have been done. Thus it is impossible for the task representing Lookup 3 to have an execution time of 20 without the Lookup 2 task also taking 20 cycles. Such correlated distributions are represented by means of a *joint probability distribution table*. Table 4.1 shows the joint probability distribution table for the lookup tasks of the IPv4 application. Note that the distribution is discrete with 6 independent scenarios, involving 2-7 memory lookups. The final lookup of the egress port (Lookup 7) needs to be performed under all these scenarios.

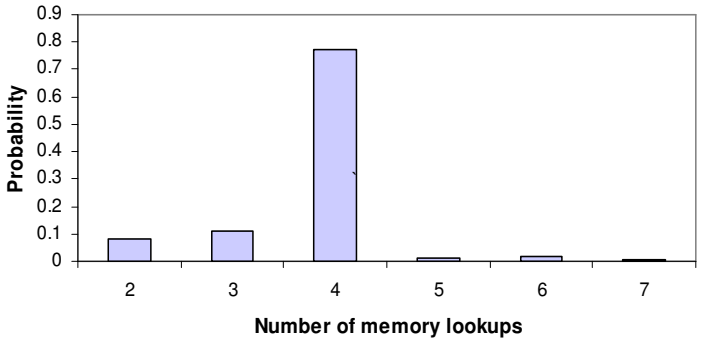


Figure 4.4: Probability distribution of the number of memory accesses for lookup

Variations due to the architecture

We implemented the IPv4 packet forwarding application using the Embedded Development Kit (EDK) from Xilinx on an Xilinx Virtex-II Pro 2VP50. We used the MicroBlaze soft processors for computation and the on-chip BlockRAM for storing the route table. We connected the processors to the BlockRAM using the OPB CoreConnect bus and the processors to each other using FSL FIFO links.

Our implementation of IPv4 on soft multiprocessors systems on FPGAs yields few sources of variations. The MicroBlaze soft processor IP is itself has an in-order 32-bit RISC engine and thus does not cause variability due to out of order executions. The remaining potential sources of variability are due to cache misses and bus arbitration. The MicroBlaze processor does have a facility to have a small data cache that can be used to cache route table entries. However, this is constructed out of the same BlockRAM that is used to store the route table. Due to the large size of our route table, we we could only afford a 8 KB data cache per processor. The measured hit rate, under the assumption that all entries in the route table are accessed equally is less than 5%. Besides, our route table is entirely present on-chip, and thus the gains from caching are minimal.

The final potential source of variations comes from the on-chip peripheral buses that transfers data to and from the route tables. These are a potential source of variability due to arbitration effects. However, our experiments showed no significant variability when up to two masters share the same bus, and a big drop-off in performance if more than two masters are present. We restricted our design space to multiprocessor systems where fewer than 2 masters share the same OPB bus, and instead used multiple OPB buses when more than two processors need to be connected to the same piece of memory.

4.1.2 H.264 video decoding on commercial multi-core platforms

H.264 or MPEG-Part 10 is an advanced video codec proposed by the ITU-T Video Coding Experts Group (VCEG) together with the ISO/IEC Moving Picture Experts Group (MPEG). The H.264 standard aims at providing high video quality at lower bit rates than previous standards, while maintaining the complexity of the implementation at practical levels.

H.264 is a block based video compression standard. Each video segment is split into frames, and each frame is further divided into macroblocks. Each macroblock represents a 16x16 pixel area in a frame. These macroblocks are encoded within the input compressed stream; and must thus be extracted from the stream. The macroblocks so decoded only contain the differences of pixel values from a predicted value. The predicted value is obtained from the decoded values of already decoded macroblocks, hence introducing data dependencies. This prediction is added to the macroblock extracted to obtain the final macroblock, which must then go through a de-blocking filter to remove edge artifacts.

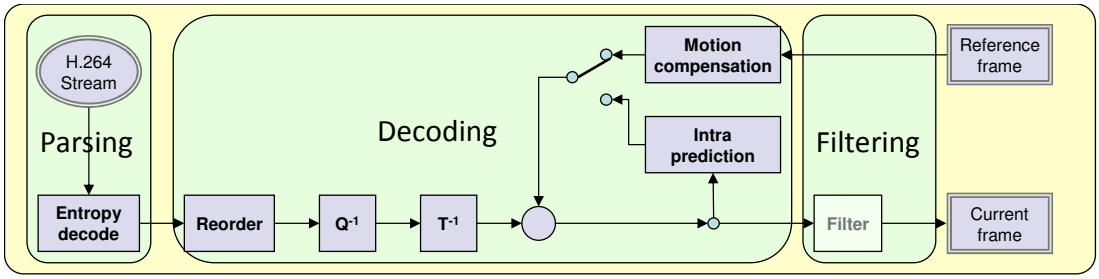


Figure 4.5: Block diagram of the H.264 decoder.

As part of the baseline profile mainly meant for mobile applications, the H.264 standard supports two types of macroblocks depending on the way in which the predicted values of the macroblocks are stored. The first of these is called Intra macroblocks or I-macroblocks. These macroblocks use the values of previously decoded macroblocks within the same frame of video to provide the predicted value. Such macroblocks rely on spatial coherence within a video frame. The other type of macroblock is the Predicted macroblock (or P-macroblocks), which use previously decoded values from previous frames (with some offsets) as the prediction. Both these types of macroblocks also have a varying amount of residual information that cannot be inferred from previous macroblocks. Figure 4.5 shows the block diagram for the H.264 decoder. The application can be broadly divided into (1) parsing: involving entropy decoding, reordering, inverse quantization and

inverse transform steps to obtain the difference macroblock values, (2) rendering: involving motion compensation for computing the predicted values of P-macroblocks and intra-prediction for computing the predicted values of I-macroblocks, and (3) the final de-blocking filter. The rendering step is the most compute intensive step in the application. We concentrate on techniques to parallelize this step.

The parallelism in the rendering stage is due to the potential for simultaneous decoding of different macroblocks in each frame. In this work, we assume that frames are decoded sequentially - each frame must complete before the next starts. In most cases, such as assumption is necessary: a P-macroblock can use decoded values from any macroblock in the previous frame, thus introducing a sequential dependency between frames. The H.264 standard allows for multiple previous frames to be accessed, all of which must be stored in a global *frame buffer*. However, a P-macroblock does not impose any ordering on the processing of macroblocks within the frame.

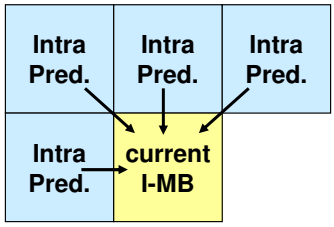


Figure 4.6: Spatial dependencies for I macroblocks in a H.264 frame.

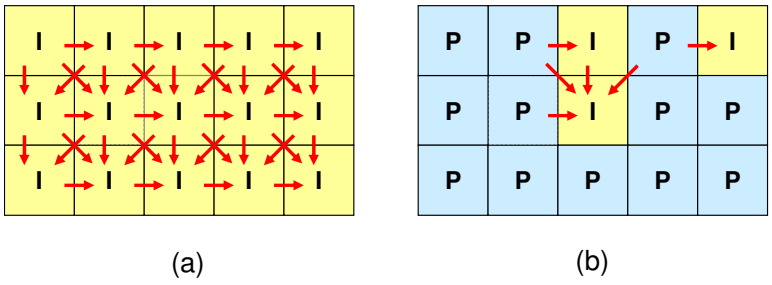


Figure 4.7: Partial task graph for (a) an I-frame and (b) a P-frame for H.264 video decoding.

In contrast, an I-macroblock does not introduce sequential dependencies between frames; however, it uses decoded values of other macroblocks within the frame. The H.264 standard imposes the restriction that an I-macroblock during intra-prediction can only predict data from the pixels to the

left, top-left, top and/or top-right relative to the current macroblocks. This introduces a dependency between macroblock executions as shown in Figure 4.6. Figure 4.7 shows a portion of the task dependency graph for an I-frame that consists entirely of I-macroblocks, and a P-frame that consists of a mixture of I- and P-macroblocks.

Variations in dependencies between macroblocks

From the above discussion, it is clear that the dependency pattern for macroblocks within a frame depends on the distribution of I- and P-macroblocks within the frame. While I-frames consisting entirely of I-macroblocks have predictable dependency patterns, the same is not true for P-frames. The exact locations of P-macroblocks in a P-frame can only be found at run-time, as it differs from frame to frame of the input video stream and is determined by the particular encoder used to encode the stream. However, by profiling the application with a set of video streams, we can collect statistical information about the probability that a macroblock will be an I- or P-macroblock. We collect this information for each macroblock across all frames of different streams and use their average as the probability of the macroblock being an I-macroblock. The probability of a macroblock being an I-macroblock determines the probability of the edges from neighboring macroblocks as per Figure 4.6.

Variations in macroblock execution times

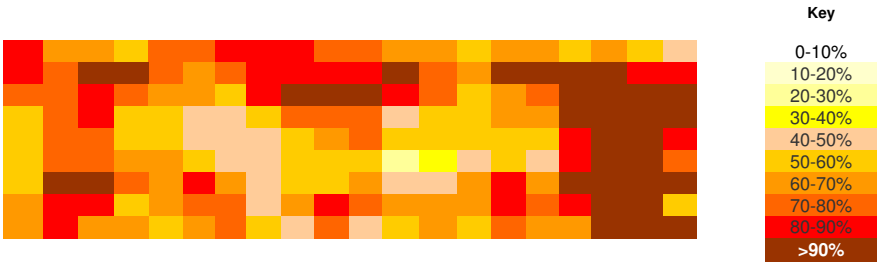


Figure 4.8: Spatial distribution of P-skip macroblocks within frames of the manitu.264 stream. Different colors encode the probabilities that particular macroblocks are P-skip macroblocks.

The variations in macroblock execution times come from three major reasons: (1) differing execution time characteristics of I-macroblocks and P-macroblocks, (2) variation in execution time among different I- and P-macroblocks due to varying amount of difference information, and (3) variations due to architecture related parameters. P-macroblocks tend to take longer than I-macroblocks

due to the need to access global frame buffer memory to obtain reference frame information. However, certain P-macroblocks are of a special kind, called P-skip macroblocks, for which there is no difference information - the macroblock is just a direct copy of the previous frame. P-skip macroblocks have low decoding times as there is no execution involved in the decoding process. Figure 4.8 shows the spatial distribution of P-skip macroblocks in the manitu.264 video stream. The color encoding represents the probabilities that particular macroblocks are P-skip macroblocks across all frames of the stream. Macroblocks towards the edges of the frame are more likely to represent background information which can be skipped and copied from previous frames. Macroblocks towards the center are much less likely to be P-skip macroblocks, and hence will take longer to execute. Similar statistics can also be collected regarding the distribution of I-macroblocks in the frame. Given these statistics, we can compute the probability of specific macroblocks being I-, P- or P-skip macroblocks.

However, the knowledge of the distribution of I-, P- and P-skip macroblocks alone is insufficient to fully characterize execution time distributions of different macroblocks. This is because the execution time of I and P macroblocks can each vary significantly. In general, the amount of difference information is indicative of the amount of work that needs to be done to reconstruct the macroblock. The combination of the distribution of types of macroblocks and the execution time variation among macroblocks of a single type gives us the full distribution of macroblock execution times.

Variations due to architecture related parameters

In addition to the above factors, the execution times of macroblocks is also affected by architecture-related parameters. Our implementation is based on the baseline profile of the H.264 decoder in C from [Fiedler and Baumgartl, 2004]. The architectural platform is a commercial 2.66 GHz Core 2 Quad platform. The main impact of the architecture comes from the out-of-order processing in the core and the effect of caches on the global frame buffer access time for P-macroblocks. The existing literature on H.264 decoding attributes most of the variation to branch misprediction penalties and not to frame buffer accesses [Hughes *et al.*, 2001] [Shih *et al.*, 2004].

Our approach to quantifying the extent of variability in macroblock execution times is through a profiling based approach. We run each video sequence through a number of times and note the execution times of each I-, P- and P-skip macroblock in each of the runs. The resulting distribution of execution times incorporates both variations due to the architecture (branch misprediction and

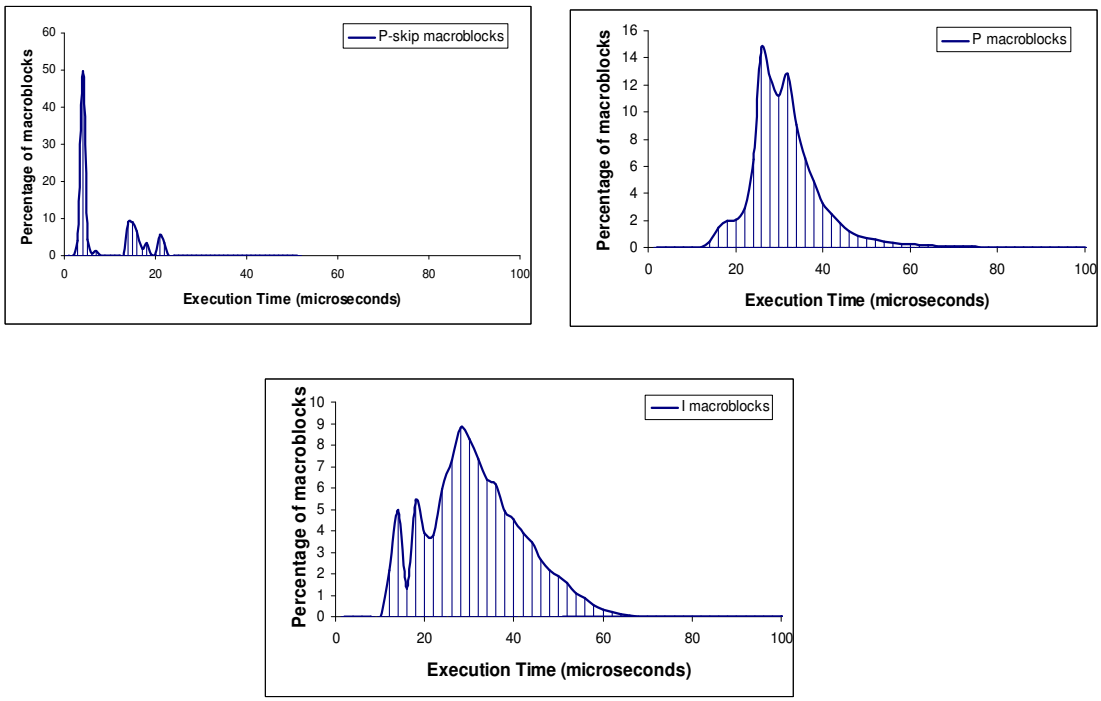


Figure 4.9: Execution time variations of P-skip, P and I macroblocks in the manitu.264 video stream.

cache effects) and application related factors. Figure 4.9 shows the distribution of execution times for I, P and P-skip macroblocks. From the figure, we can see that the execution time of different P- and I-macroblocks can differ by close to an order of magnitude. It is thus extremely important to capture this variation in the form of a statistical model.

4.2 Statistical Models

In the following sections, we only focus on developing models and methods that make scheduling decisions statically in the presence of variations. Even when scheduling is done dynamically, statistical models are useful to capture unpredictable sources of variations such as cache misses and bus arbitration effects. Since our modeling technique is based on profiling the application through traces, such unpredictable sources of variations can be transparently incorporated. As in the static delay model, we first describe the application model that describes the concurrent tasks and the dependencies, the architecture model that exposes the parallelism in the architecture, and then the performance model that estimates the performance of a task on a processing element.

4.2.1 Application Task Graph

We introduced the task graph model for static scheduling in Chapter 2. The task graph is a directed acyclic graph (DAG) $G = (V, E)$ with the vertexes V representing the set of tasks and the edges E representing task dependencies and data transfers. Each task represents a set of instructions that must be executed without pre-emption on a single processor. The dependence edges enforce the constraint that tasks can start only after all their predecessors have finished executing. The task graph is a specialization of the static dataflow (SDF) model of computation. In particular, it represents an acyclic homogeneous data flow graph, where each task is executed exactly once in a single run of the application and the order of execution respects the dependencies between tasks.

The static task graph model is directly applicable to the statistical scheduling problem if there is no variation in task dependencies. The IPv4 packet forwarding application described in Section 4.1.1 is an example of with deterministic dependencies. Figure 4.1 shows the task graph of the IPv4 packet forwarding application. All dependencies between tasks are known at compile time. For instance, while "Verify time-to-live" and "Verify checksum" can be executed in parallel, the "Update checksum" task can only start after both the verify tasks complete. Another example of this is in decoding the I-frames in the H.264 video decoding application. The task graph in Figure 4.7(a) represents the decoding of a single I-frame of H.264, with each task decoding an I-macroblock. Each I-macroblock depends on the macroblocks to its left, top, top-left and top-right (Figure 4.6).

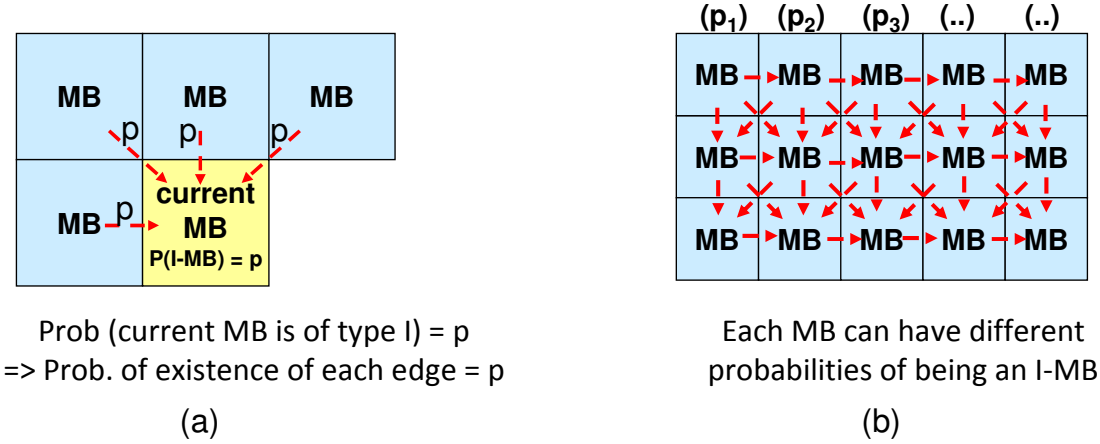


Figure 4.10: (a) Probabilistic dependencies for a single task in a P-frame of a H.264 decoding task graph. (b) shows a partial task graph for P-frames with more than one task.

Often, some or all of the dependencies between tasks cannot be completely determined at

compile time. Such a situation may occur because the dependencies are input dependent, as is the case in the P-frames of the H.264 video decoding application. In this case, we modify the task graph to add edge probability weights that indicate the statistical probability of the edge being present. Figure 4.10(a) shows the dependencies for decoding a single macroblock of a P-frame in H.264 decoding. While I-macroblocks depend on neighboring macroblocks, P-macroblocks have no dependencies on macroblocks in the same frame. Thus the probability of a task representing an I-macroblock determines the probability of the edges from tasks to its left, top, top-left and top-right. Different macroblocks have different probabilities of being an I-macroblock. Thus the probabilities on different edges can be different. A portion of the task graph for a P-frame of H.264 is shown in Figure 4.10(b).

Formally, a statistical task graph is a DAG $G = (V, E, L)$ with V and E representing the set of tasks and (probabilistic) edges, and $L : E \rightarrow (0, 1]$ representing the non-zero probability (likelihood) of each edge dependency. Edges with zero probability need not be represented in the model. The presence of probabilistic edges cannot be easily handled in a static task graph model. Each edge in a static model must either be present or not; and we have to be conservative in our decisions so as to produce a valid schedule. Thus the static task graph would assume each edge with non-zero probability to be present. As we shall see in Section 4.3.4, such worst-case assumptions lead to a significant overestimation in application finish time.

4.2.2 Architecture Model

In Chapter 2, we introduced the (P, C) model for multiprocessor architectures. In the (P, C) model, the multiprocessor network is modeled by a set of processors P connected by a set of direct point-to-point links. The links are represented by $C \subseteq P \times P$, the set of processors that are connected to each other (Chapter 2). We use the same model for statistical scheduling as well.

4.2.3 Performance Model

In the previous sections, we developed a representation of the concurrency in the application and the parallelism in the architecture. In order to determine the performance of a mapping of the application to the architecture, it is important to annotate the task graph with a performance model. In Chapter 2, we discussed the static delay model (G, w, c) where $G = (V, E)$ denotes the static task graph, $w : V \times P \rightarrow \Re^+$ with $w(v, p)$ being a real valued constant denoting the execution time of task v on processor p, and $c : E \times C \rightarrow \Re^+$ with $c(e, (p_1, p_2))$ being a constant denoting the

communication latency of the data transferred along edge e on the link connecting p_1 and p_2..

However, as discussed in Section 4.1, the execution times of tasks can vary due to input dependent control flow or due to the effect of architectural features such as caches. The communication latency between tasks can also exhibit variance if the communication link is a bus with variance due to bus arbitration, or is a packet-based network where packets may be retransmitted after a loss. In such cases, the model cannot use real-valued constants to represent weights of tasks and edges. The weights $w(v, p)$ and $c(e, (p_1, p_2))$ are instead represented as random variables that can take values in the set of real numbers $\Re$. The values taken by these random variables are encoded through their probability distributions.

There have been previous attempts at characterizing execution times of tasks [van Gemund, 1996] [Gautama and van Gemund, 2000] [Iverson *et al.*, 1999]. These can be primarily classified into analytical and simulation based approaches. Analytical approaches build up the variability of a task from the variability of each statement of code and the control structure of the code [Iverson *et al.*, 1999] [van Gemund, 1996]. These can be accurate for variations arising due to input dependent execution traces. However, these are usually difficult to build, and cannot easily account for random sources of variation such as bus arbitration effects. Simulation based approaches execute the code with different inputs on a simulator of the platform or the real platform itself and obtain the distribution of real execution times [Gautama and van Gemund, 2000]. Such an approach is simpler but assumes access to the final platform. Frameworks consisting of a combination of the two approaches have also been proposed [Yang *et al.*, 1993].

In this dissertation, we use a simulation-based approach to characterizing the runtimes of different tasks. There are two types of variations we wish to capture: (1) variations in runtime due to different inputs exciting different execution traces in the task and (2) variations in the runtime with the same input across many runs due to cache effects or bus arbitration. To capture the first effect, we simulate the runtimes of each task in the task graph with different inputs. To capture the second effect, we simulate the runtimes of different tasks in the task graph a number of times while other tasks are executing on different processors.

Representing distributions

The distribution of a random variable is commonly represented by means of its *probability density function* (p.d.f). The probability density function of a random variable is a function whose integral over a given interval gives the probability that the values of the random variable fall within

that interval. In case the random variable is discrete rather than continuous, the integral is replaced with a summation over the discrete values that the random variable can take within the considered interval.

In this work, we discretize all continuous random variables by binning the range of values that the random variable takes and using the midpoint of each bin to represent all the values contained in that bin. This enables us to uniformly represent all distributions by means of a *probability distribution table*. Such a table would contain entries of the form (variable range, probability), and represents the probability that the random variable is in the specified range.

Assumption of task independence

The delay model for static performance estimation assumes that task execution times and communication delays are independent. However, task execution times in many applications are correlated. The runtimes of the lookup tasks in the IPv4 application are shown to be correlated in Section 4.1.1. For correlated random variables, individual probability distributions cannot be used to represent distributions; we instead need to store the *joint probability distribution tables* of all the correlated variables. Table 4.1 shows the joint probability distribution table of the lookup tasks in IPv4 forwarding. The entries here are of the form ($range_1$, $range_2$, ... , $range_n$, probability), which stores the joint probability that the value of random variable 1 lies in $range_1$, the value of random variable 2 lies in $range_2$ and so on.

It may so happen that only certain sets of tasks have correlated execution times, while others do not. In fact, the tasks in the IPv4 task graph of Figure 4.1 that are not lookups are independent (and in fact have constant execution times). In this case, the joint probability distribution table only needs to contain the correlated variables. Thus for our IPv4 application, we only require the joint probability distribution of Table 4.1 and constant execution times for other tasks. The task graph for the IPv4 application with the annotated probability distributions is shown in Figure 4.11.

For the sake of notational convenience, it is possible to consider constant static values as random variables that take only one value. The probability distribution table then contains exactly one entry: (value, 1.0). This indicates that the random variable only takes one value with a probability of 1.0.

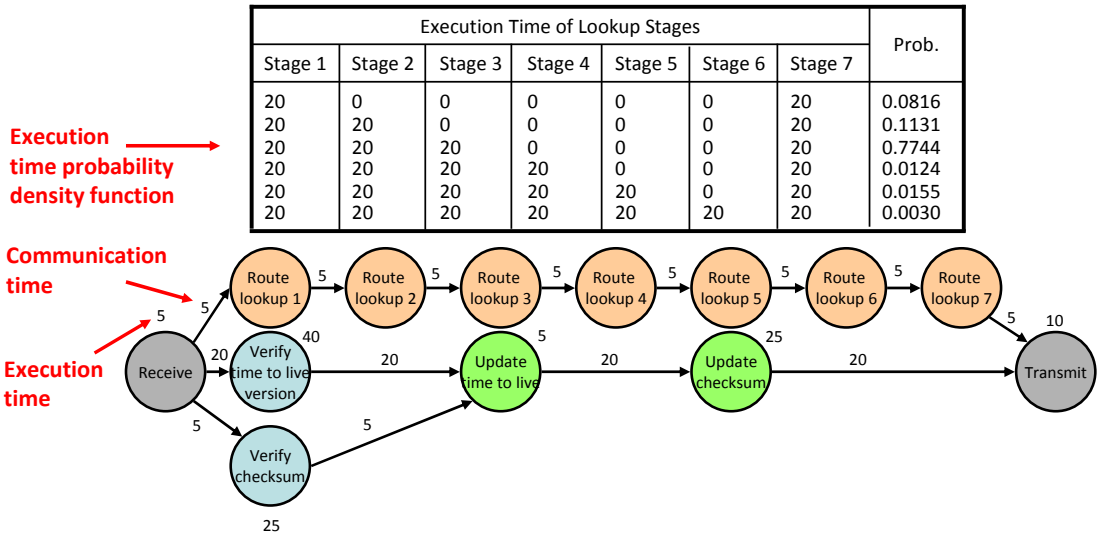


Figure 4.11: Task graph for IPv4 packet forwarding annotated with the performance model.

4.2.4 Optimization Problem

The static scheduling optimization problem as described in Chapter 2 is to find (1) an allocation of tasks to processors and (2) a start time for each task while respecting task dependency constraints in order to optimize for the end-to-end finish time for the application or makespan (Section 2.1.4). While the makespan for a static scheduling algorithm can be computed as a single number, this is no longer the case for statistical scheduling. Since the execution and communication times are now distributions, the makespan of a valid schedule is a distribution as well.

The focus of this work is on soft real-time applications where not all the inputs in the input stream need be processed. Such applications instead have a fixed statistical requirement on the percentage of all inputs that must be processed. For instance, an IPv4 packet forwarding algorithm may only require that 99% of its input packets are processed. For such applications, the performance of the application is not accurately measured by the worst or average case value of the makespan distribution, but rather by a fixed *percentile* of the makespan distribution. The η'th percentile of a distribution ($0 \leq \eta \leq 100$) is defined as the value below which $\eta\%$ of the observations of the distribution fall. For the IPv4 forwarder that requires 99% of all inputs to be processed, the performance of the application is measured as the 99^{th} percentile of the makespan distribution of the forwarder. The required guarantee η on the percentage of inputs to be processed depends on the application and the quality of service that is needed. It is common for IP forwarding to have different quality

of service requirements for different classes of traffic. In such a context, the value of η is often an additional parameter for performance analysis.

A scheduling algorithm based on this metric of performance analysis must then take in η as a parameter. In this work, we consider the required percentile η to be one of the inputs to our scheduler. The objective of the scheduling problem is to then find an allocation and schedule to minimize the η^{th} percentile of the makespan distribution. In order to solve this optimization problem, we must first be able to evaluate the objective function for a given allocation and schedule. The problem of computing the η^{th} percentile of the makespan given a valid allocation and schedule is called the *statistical analysis* problem.

4.3 Statistical Performance Analysis

In this section, we consider the problem of computing a required percentile of the makespan given a valid allocation and schedule. We first proceed by defining a valid allocation and schedule.

4.3.1 Valid allocation and schedule

Given a statistical task graph $G = (V, E, L)$, an architecture model $H = (P, C)$, and a performance model (w, c), we define a valid *allocation* $A : V \rightarrow P$ that assigns every task in V to a single processor in P. As in Chapter 2, we do not consider the more general case where a single task is broken up into smaller tasks. Given a particular A, the communication delay between tasks v_1 and v_2 is defined as $c(e, (A(v_1), A(v_2)))$, where $e = (v_1, v_2)$ represents the edge between tasks V_1 and v_2, and $(A(v_1), A(v_2))$ represents the communication link between processors $A(v_1)$ and $A(v_2)$. A *schedule* is a function $S : V \rightarrow D$, which assigns a random variable $d \in D$ as the start time of the task. The random variable d only takes values from the non-negative real numbers, and is represented by its probability density function p_d where for each $x \in \Re^+$, $p_d(x) = Prob(d = x)$. For the schedule to be valid, it must satisfy two constraints:

$\forall (v_1, v_2) \in E$ (dependence constraints),

$$(a)\ S(v_2) \geq \begin{cases} S(v_1) + w(v_1) + c((v_1, v_2), (A(v_1), A(v_2))) & \text{with probability } L(v_1, v_2), \\ 0 & \text{with probability } 1 - L(v_1, v_2). \end{cases}$$

$\forall v_1, v_2 \in V, v_1 \neq v_2$ (ordering constraints),

$$(b)\ A(v_1) = A(v_2) \Rightarrow S(v_1) \geq S(v_2) + w(v_2) \vee S(v_2) \geq S(v_1) + w(v_1)$$

Constraint (a) enforces that the task dependencies are met probabilistically: a task can start only after all its predecessors complete. However, this constraint only holds with the probability of the edge dependency $L(v_1, v_2)$. If the dependence does not exist (probability $1 - L(v_1, v_2)$, then there is no constraint imposed on the start time of task v_2, represented as $S(v_2) \geq 0$. We can interpret this constraint as follows: at run time, given a stream of inputs, a certain fraction of those inputs will result in a particular edge dependency (v_1, v_2) begin present. For those inputs, a run time system delays the execution of task v_2 until task v_1 is complete. For other inputs where the dependency does not exist, task v_2 is allowed to run at any point of time. The probabilistic constraint captures both these scenarios and computes the probability distribution of the start time of task v_2.

We can rewrite constraint (a) as:

$$(a')\ S(v_2) \geq L(v_1, v_2) * (S(v_1) + w(v_1) + c((v_1, v_2), (A(v_1), A(v_2))))$$

where the operation $k * D$, k being a constant ($0 \leq k \leq 1$) and D being a distribution is defined by the following procedure: multiply each probability in the probability distribution table corresponding to D by k. If distribution D already takes the value 0 with some non-zero probability $p_D(0)$, then add $(1-k)$ to $p_D(0)$; else add a new entry of 0 to the table with probability = $(1-k)$. We note that (a') is merely a short form representation for the operation in (a).

Constraint (b), as in the static case enforces a total ordering of the set of all tasks assigned to a processor. There is no probabilistic nature to these inequalities, since they arise due to the hard constraint that tasks allocated to a single processor should not overlap. It is possible that two tasks v_1 and v_2 are assigned to the same processor may already have a dependence edge (v_1, v_2) between them in G. In such a case, constraint (b) implies that $S(v_2) \geq S(v_1) + w(v_1)$, which is stronger than constraint (a) for that edge. Hence constraint (a) is unnecessary for such pairs of tasks.

4.3.2 Formulation of the performance analysis problem

The *makespan* of a valid allocation and schedule is defined as the η^{th} percentile of $\max_{v \in V} S(v) + w(v, A(v))$ for a given η. The performance analysis problem comes down to computing this quantity. In the following sections, we consider the problem of obtaining the makespan distribution. Given the makespan distribution, we can easily read off the required percentile as the makespan.

Given the task graph $G(V, E, L)$, an allocation A and a schedule S, we can capture the schedule graphically by by adding additional edges to the task graph. Define

$$E'' = \{(v_1, v_2) \notin E | A(v_1) = A(v_2) \wedge S(v_2) \geq S(v_1) + w(v_1)\}$$

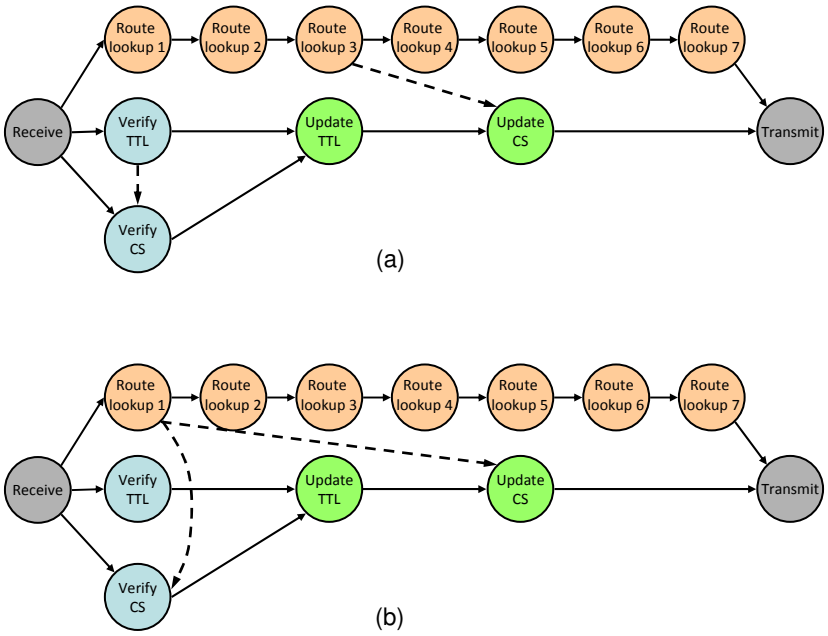


Figure 4.12: Two valid schedules for the task graph in Figure 4.1. The dashed edges represent the ordering edges.

to be the set of *ordering* edges that define the total order in which tasks allocated to the same processor execute. Define $G' = (V, E', L')$ where $E' = E \cup E''$ and

$$L'(e) = \begin{cases} 1.0 & \text{if } e \in E'' \\ L(e) & \text{if } e \notin E'' \wedge e \in E'' \end{cases}$$

G' is the graph with the ordering edges added. The probability of each of the ordering edges is set to 1. In case there is an edge e that is both a dependence and ordering edge, then the probability of existence of the edge is replaced by 1. G' remains a directed acyclic graph for a valid schedule. Analysis of the schedule is then a longest path computation on G', which can be computed with a breadth-first traversal of G'. Figure 4.12 shows two valid schedules for the IPv4 forwarding application. The dashed edges represent the ordering edges. All edges in the graph have a probability of 1.

The challenge in computing the longest path comes from the fact that each node execution time and edge communication time is an arbitrary random distribution expressed as an (individual or joint) probability distribution table. Furthermore, the dependence edges are probabilistic as well, requiring manipulation of the probability distribution tables. When traversing each node v of the

graph in a breadth-first manner, the following operation needs to be performed:

$$\max_{u:(u,v)\in E} L(u,v) * (S(u) + w(u) + c((u,v),(A(u),A(v))))$$

There are thus three types of operations involved in the longest path computation: (1) *sum* of execution and communication times along a path of the graph and (2) *max* of arrival times at nodes with more than one incoming edge and (3) performing the $*$ operation. These operations must be carried out statistically, taking into account the distributions and potential correlations between different distributions.

4.3.3 Types of performance analysis

The techniques used in solving the statistical performance analysis problem depend on the statistical model used. In this section, we shall describe techniques to analyze the general statistical model that we described in the previous section. We justify our need for a general model in Section 4.4.

Exact Performance Analysis

An exact analysis aims at computing the exact distribution for the longest path of the scheduled task graph G' described in Section 4.3.2. The key components to the analysis are computing the sum, max and $*$ operations on a set of random variables. Of these, the third operation involves a direct manipulation of the probability distribution tables as already described in Section 4.3.2. It then remains to compute the sum and max of arbitrary random distributions.

We first assume that A and B are independent distributions. Let $C = A + B$ be the sum of these two random variables. For this case, the probability distribution of C (p_C) is known in the literature to be the convolution of the probability distributions of A and B and is represented as $p_C = p_A \circ p_B$ [Freedman *et al.*, 2007]. p_C is computed as:

$$p_C(x) = p_A \circ p_B = \sum_{y \in R_A} p_A(y) \cdot p_B(x - y)$$

The sum of more than two distributions $S_n = X_1 + X_2 + \cdots X_n$ can be computed using associativity of the summation operator as $S_n = S_{n-1} + X_n$, where $s_{n-1} = X_1 + x_2 + \cdots X_{n-1}$.

The probability density function of the maximum of random variables is hard to compute directly. Instead, we can compute the cumulative probability distribution function, defined as

$P_A(x) = \sum_{y \leq x} p_A(y)$. Given distributions A and B with cumulative probability distributions P_A and P_B, we can compute the cumulative probability distribution of $D = max(A, B)$, P_D as:

$$P_D(x) = P_A(x) \cdot P_B(x)$$

The probability density function $p_D(x)$ can be computed from P_D by successive differences between adjacent P_D values.

However, the assumption of independence of variables is inaccurate. There are many sources of correlation between these variables including: (1) the individual task execution times or communication times may in themselves be correlated due to the nature of the application (as in IPv4 packet forwarding), or (2) even if individual task execution times are independent, the execution times of the sums of task execution times along different paths may be correlated. As a simple example, the execution times of paths $A \rightarrow B$ and $A \rightarrow C$ are correlated because the two paths share a common task A. If both these paths are inputs to task D, then we cannot use a simple max computation as outlined above to compute the start time of D.

The only way to compute exact sum's and max's in the presence of arbitrary correlations is a brute-force approach of trying every possible combination of values of each of the random distributions involved. For instance, to compute $D = \max(A+B, A+C)$, we need to consider each triplet of values of ($a \in A$, $b \in B$ and $c \in C$), and compute $\max(a+b, a+c)$ for each such triplet, with the corresponding probability of $P(A = a \wedge B = b \wedge C = c)$. If each of A, B and C can take k values, then this computation involves order of k^3 computations. Each of these computed values can be unique: the probability density function of D can contain up to k^3 values. These values needs to be propagated on to the successors of D in the task graph, and can be involved in further correlations. In the worst case, each combination of the set of all random variables in the task graph needs to be considered for computing the makespan distribution. In general, in order to compute the longest path of a graph with n variables each of which can take k distinct values, we need to perform order of k^n computations, which is exponential in n. Thus computing the exact makespan distribution is usually intractable. This motivates the use of approximations for computing the performance of the application. We now describe one such approximation based on a commonly used technique called Monte Carlo statistical analysis.

Monte Carlo Analysis

Monte Carlo methods rely on repeated random sampling to perform statistical analysis. Monte Carlo simulations are typically used when exact approaches are impossible or infeasible due to com-

putational requirements [Hubbard, 2007]. Each random sample in a Monte Carlo simulation gives us a particular value that the required distribution can take. As the number of samples increases, the frequency of these values approximates the required distribution. Monte Carlo methods rely on the laws of large numbers to ensure that the estimate converges to the correct value as the number of draws increases. Monte Carlo simulations have been used in a number of applications related to insurance risk analysis [Vose, 2008], computational finance [Glasserman, 2004], computational physics and computational mathematics among many others.

A general Monte Carlo simulation has the following steps:

1. Define the domain of possible inputs
2. Generate random inputs from the domain according to the given probability distributions and perform a deterministic computation on these inputs
3. Combine the computational results of all deterministic runs to give the aggregate result

We now discuss the application of Monte Carlo simulations to computing the makespan distribution of a scheduled task graph. We begin by noting that the domain of possible inputs is defined in the statistical model. There is a first set of random variables corresponding to the edge probabilities L. Further, under the assumption that each probability distribution table (joint or individual) is independent of all other tables, each of these distributions represents an independent random variable with the domain being the set of entries in the table. Each individual probability distribution table represents one execution or communication time variable; while a joint table corresponds to more than one original execution or communication time variables. The assumption of independence of probability distributions may not hold in cases where a subset of random variables in two joint probability distributions are shared. In such cases, we will combine the two tables with a single joint probability distribution, which will then satisfy the assumption.

Each independent random variable corresponding to an edge probability or a probability distribution table must be sampled according to the respective distributions. For each edge e with a probability of existence $l(e)$, this is achieved by choosing a random number r uniformly in the range [0,1], and choosing the edge to be present if $r < l(e)$. A similar scheme is used to sample the probability distribution tables of execution and communication times. For a table with two entries with probabilities 0.6 and 0.4 respectively, we must ensure that the first entry is chosen in 60% of all samples and the second is chosen 40% of all samples. A common method of sampling probability distribution tables is by generating uniformly distributed random numbers in the range [0,1] and us-

ing the inverse of the cumulative probability distribution function to obtain the sample. In the above example, we would generate uniform random numbers in the range [0,1]. The cumulative probability distribution table for this example is $F(x) = 0.6,\ x \leq entry_1,\ 1.0$ otherwise. The inverse of this function is $F^{-1}(y) = entry_1,\ y \leq 0.6,\ entry_2$ otherwise. We would choose the first entry whenever the uniformly generated random number is ≤ 0.6 and use the second entry otherwise. We can easily see that this would gives us the right percentage of both entries with sufficient samples. A similar procedure is used to sample joint probability distribution tables, except that each sample corresponds to more than one (dependent) random variable value.

After samples have been generated for all tables, we take the first sample of all independent random variables (one for each table) and use the deterministic value obtained for all the execution and communication times (the dependent random variables) to perform a deterministic analysis of makespan. Deterministic analysis boils down to a linear time longest path computation on the scheduled DAG. This procedure is then repeated for other samples, yielding a sequence of makespan values. This is then aggregated into a probability distribution function for the makespan.

The main advantage of using Monte Carlo techniques for statistical analysis is that it is a very general method that can be used with any type of random distributions. Further, the method inherently takes into account any correlations between the random variables. This generality does come at a cost: Monte Carlo simulations rely on the law of large numbers to achieve convergence to the exact distribution. It is typical to require a large number of iterations ($\sim$1000) to reduce the error from the exact distribution to acceptable levels. It has been shown that the error of Monte Carlo simulations from the exact solution $\epsilon \propto \frac{c}{\sqrt{n}}$, where n is the number of Monte Carlo simulations and c is a constant depending on the nature of the distributions involved. This proportionality has two important consequences: (1) the rate of convergence is independent of the number of variables, except insofar as it affects the constant term (typically small), and (2) there is a law of diminishing returns with respect to errors - in order to achieve one order of magnitude improvement in error, we require to perform 100 times as many iterations.

In order to judge the accuracy of Monte Carlo for our problem, we compare Monte Carlo makespan distributions to that obtained from the exact performance analysis procedure described earlier in this section. A common metric used to judge the difference between two discrete distributions is the *chi-squared difference*, defined as

$$\chi^2 = \sum_{i=1}^{k} (p_{MC}(i) - p_{exact}(i))^2 / p_{exact}(i)$$

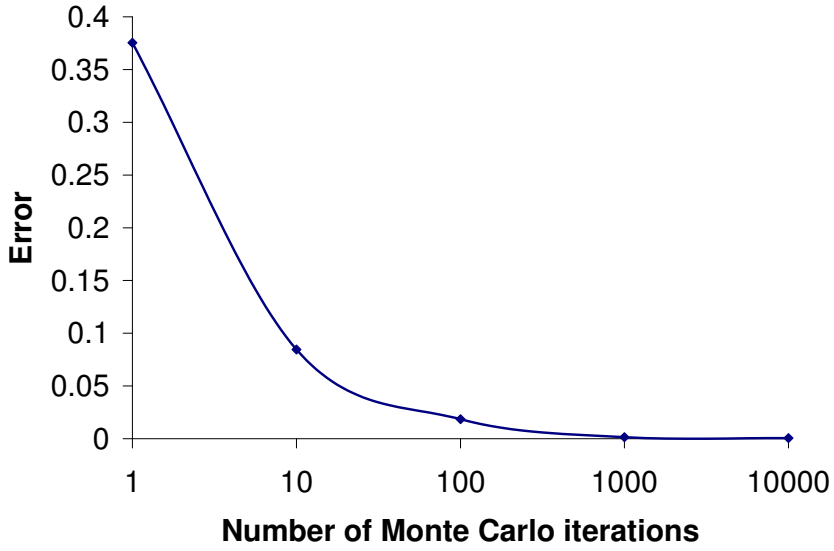


Figure 4.13: χ^2 error for Monte Carlo analysis with varying numbers of Monte Carlo iterations versus exact analysis for a 4-port IP forwarder.

where p_{MC} and p_{exact} denote the probability density functions of the makespan obtained by Monte Carlo analysis and exact analysis respectively, and i ranges over the set of discrete values that are taken by these probability distributions. Figure 4.13 shows the χ^2 error of the Monte Carlo simulations versus exact analysis for an average of 10 schedules for a 4-port IP forwarder (the task graph of Figure 4.1 was replicated four times). The figure indicates that there is not too much improvement in error to be had after about 1000 iterations. In the rest of this dissertation, we use Monte Carlo with 1000 iterations for analysis.

Deterministic Scenario based analysis

A deterministic approach to analyzing statistical task graphs is to use corner-case or common-case estimates on each random variable present in the graph. Since we are usually interested in solving the analysis problem for a high value of η (the input percentile) to provide high guarantees on makespan, a typical approach is to use worst-case estimates on all random variables. In particular, we replace the statistical task execution and communication times by their worst-case values. We also replace the statistical probability of each dependence edge with a deterministic probability of 1.0. We then perform a deterministic analysis (Section 2.2) on this deterministic task graph. This method assumes that all individual random variables will simultaneously attain their worst-case values. Consequently, this method is conservative and produces an upper bound to the statistical

analysis result.

Another deterministic approach is to use the most dominant execution trace to provide deterministic values for the random variables. Such a scheme approximate individual execution and communication times by their most common values, and retains only those dependence edges have a probability of existence greater than 0.5. All ordering edges have a probability of existence greater than 0.5 and will be retained. Of course, the result of such an analysis will not result in a valid set of start times for tasks during the actual application execution. It is expected that certain tasks (at least for some inputs) will be scheduled to start before their inputs arrive. It is important to realize, however, that the intent of statistical analysis is only as a mechanism for comparing different schedules. Using the makespan under the common-case assumption does serve this purpose. During the actual application execution, the start times of tasks are ignored; only the ordering of tasks is retained. We shall describe this mechanism later in Section 5.2.3 of Chapter 5. Using the common-case estimates for task dependencies and execution and communication times does give a good estimate of the mean makespan of the application, but does not accurately measure high percentiles of the makespan. In general, common-case analysis yields a heavy underestimate of the actual makespan.

An extension to using the most common case behavior is to use a set of most likely scenarios. Such an approach has been proposed in [Gheorghita *et al.*, 2008]. The idea here is that schedules can be found for each individual scenario and a run-time system dynamically chooses between such scenarios depending on application inputs. While such techniques are useful if there is a small set of scenarios that can be pre-characterized, it is in general hard to obtain a comprehensive set of such scenarios. Moreover, the effects of architecture-related variations are not easily captured through this method. Our statistical analysis based technique relies entirely on simulations to obtain variations, and does not require pre-characterization into scenarios.

4.3.4 Comparison of Statistical to Static Analysis

As outlined in the previous section, there are different analysis techniques that can be used to compute makespan in a statistical context: exact analysis, Monte Carlo analysis and deterministic analysis. These techniques trade-off performance for accuracy of analysis. While exact techniques are the best in terms of accuracy, they rapidly become computationally infeasible for task graphs with more than a few 10's of independent random variables. The accuracy-runtime tradeoff can be tuned in Monte Carlo analysis by adjusting the number of Monte Carlo iterations as shown in Section 4.3.3. Monte Carlo techniques provide good accuracy at reasonable runtimes when we have

only a few hundred random variables. Deterministic analysis can be done very efficiently, but is inaccurate. The choice of using deterministic or Monte Carlo techniques depends on the extent of the inaccuracy of deterministic analysis and how it affects the quality of the schedule generated. In this section, we compare the accuracy of statistical Monte Carlo analysis to deterministic worst-case and average-case analysis.

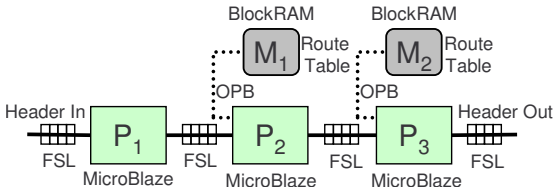
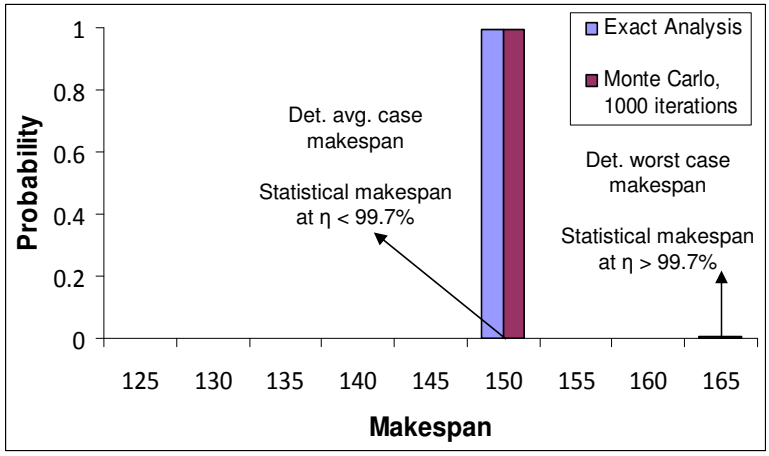


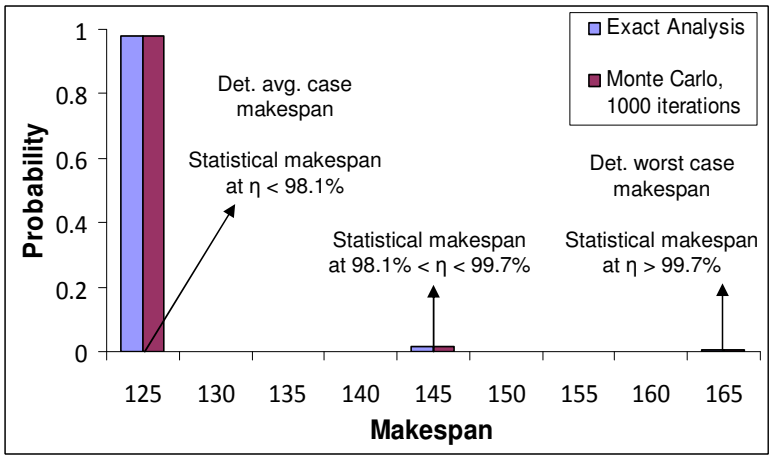
Figure 4.14: Architecture model of a soft multiprocessor composed of an array of 3 processors.

Consider the task graph for IPv4 packet forwarding in Figure 4.11 that has been annotated with the performance model. We consider possible mappings of this application onto the architecture model in Figure 4.14. This architecture model is an abstraction of a soft multiprocessor system with three MicroBlaze processors connected in a pipeline using FIFO channels, with processors 2 and 3 connected to the on-chip route table.

For the two schedules shown in Figure 4.12, we run 1000 Monte Carlo simulations of the scheduled graph to obtain the Monte Carlo makespan distribution. For η = 99%, we compare the 99^{th} percentile of the makespan to the deterministic estimates obtained by individually raising each task execution time to its worst-case estimate. The results of the comparison of Monte Carlo to the deterministic estimates is shown in Figure 4.15. For comparison, the exact makespan distribution is also shown in the same figure. From Fig. 4.15, we can conclude that Monte Carlo analysis is very accurate with respect to exact analysis. In contrast, the deterministic worst-case analysis overestimates the Monte Carlo makespan at η = 99% in both schedules (a) and (b). Further, the worst-case estimate does not overestimate both schedules evenly: it is more accurate for schedule (a) than schedule (b). A scheduling technique based on worst-case estimates would conclude that both schedules (a) and (b) have a makespan of 165, and are therefore equally good. However, the Monte Carlo simulations clearly indicate that schedule (b) is better than (a) at $\eta < 99.7\%$, with a makespan of 145 for schedule (b) rather than 150 for schedule (a). In general, worst-case estimates can yield unpredictable accuracy in makespan estimates, thereby misleading a scheduling algorithm into making wrong scheduling decisions.



(a)

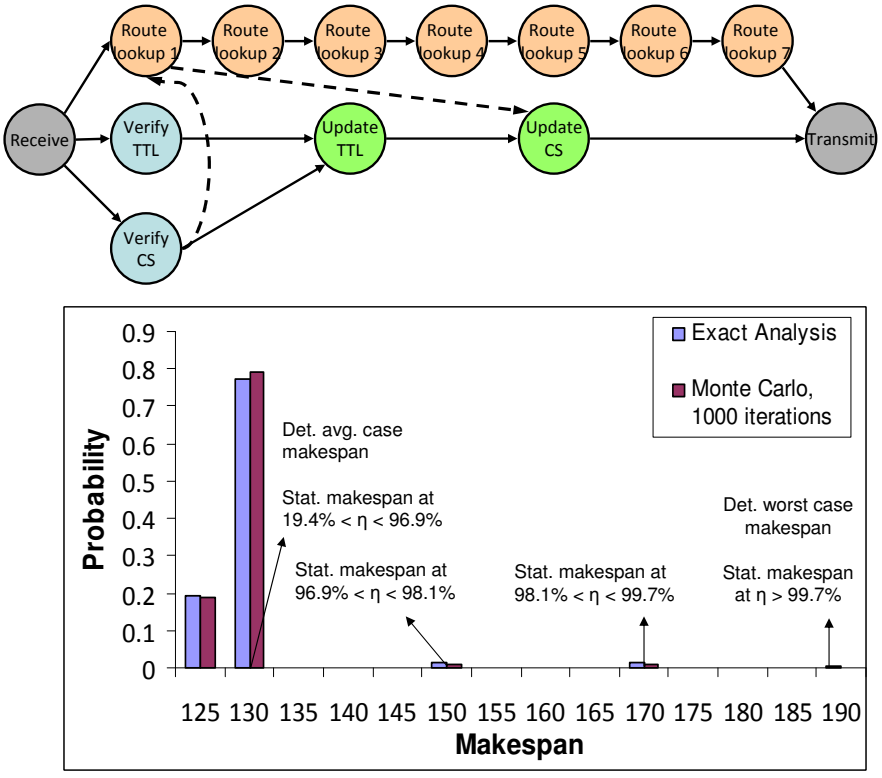


(b)

Figure 4.15: Analysis results for schedules (a) and (b) in Figure 4.12.

Another drawback of deterministic analysis is that the worst-case estimates are not sensitive to the required percentile, while the actual makespan depends on the required percentile. From Figure 4.15, we can also obtain the 95% percentile of the makespan distributions of the two schedules (which can be obtained by merely reading off a different percentile, once Monte Carlo analysis is done). We find that at a percentile of 95%, it becomes even more important to choose schedule (b) over (a): the makespan of schedule (b) is now 125, compared to 150 for schedule (a). However, the deterministic analysis is unaware of the percentile and still considers the two schedules to

be equivalent. In general, the inaccuracy of deterministic worst-case analysis becomes even more pronounced at lower percentiles, as we move away from the worst-case assumption.



(c)

Figure 4.16: A third schedule for IPv4 packet forwarding and the corresponding analysis result.

Using the most common execution times for each task (for the task graph in Fig 4.11, this would correspond to the third row of the probability distribution table with 4 active lookup stages) is an alternative to worst-case analysis. However, considering the common case does not improve the accuracy of the analysis. Figure 4.15 also shows the makespan estimates obtained by common-case analysis for schedules (a) and (b). We can see that the analysis significantly underestimates the makespan in both schedules at high percentiles. Common-case analysis also has the potential to mislead scheduling algorithms. For instance, consider a third schedule (c) for IPv4 forwarding and the corresponding analysis result in Figure 4.16. Common-case analysis judges this schedule (c) to have a makespan of 130 and thus better than schedule (a) with a makespan of 150. However, from the Monte Carlo analysis, we can see that at $\eta = 99\%$, schedule (a) is actually significantly better

than schedule (c).

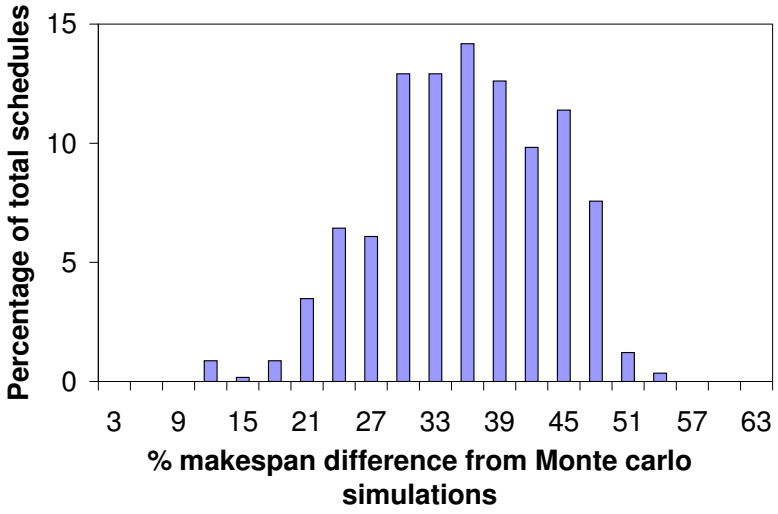


Figure 4.17: Histogram of the percentage differences between worst-case deterministic analysis and Monte Carlo analysis for a set of 1000 schedules for H.264 decoding on manitu.264. The deterministic analysis is uneven in its estimation of makespan.

These results also hold for other applications as well. In particular, worst-case overestimates the actual makespan and common-case analysis typically underestimates it; moreover, they do so unevenly. Figure 4.17 shows a histogram of the percentage differences between deterministic worst-case estimates and Monte Carlo analysis for a set of 1000 different schedules in a frame of H.264 decoding for the manitu.264 video stream. We use the execution time distributions as shown in Figure 4.9. There is a large spread in the accuracy of the deterministic worst-case estimate with respect to Monte Carlo values. Any optimization method based on this analysis is thus sub-optimal.

4.4 The need for generalized statistical models and analysis

We have so far described a very general method of representing variations based on joint and independent probability distribution tables. Statistical models have been previously used in describing statistical variations in gate delays in circuits for timing analysis [Orshansky and Keutzer, 2002][Visweswariah *et al.*, 2006][Najm and Menezes, 2004], representing activity time distributions in project management using the Program Evaluation and Review Technique (PERT) [Kerzner, 2003] [Klastorin, 2003] and financial modeling for risk analysis [Ruppert, 2006][Lai and Xing, 2008]. The approach used in these cases have been to model the statistical variations in the form

of specialized distributions such as normal distributions in circuit analysis [Visweswariah *et al.*, 2006][Orshansky and Keutzer, 2002], or beta [Gupta and Nadarajah, 2004] or triangular distributions [van Dorp and Kotz, 2002] in project management. Using specialized distributions can enable analytical approaches to analysis [Visweswariah *et al.*, 2006] [Orshansky and Keutzer, 2002] or analysis based on table lookups [Sinha *et al.*, 2007]. The key feature that enables the normality assumption in circuit analysis is the central limit theorem [Rice, 1994], which posits that the sum of a large number of random variables is approximately normally distributed. The number of random variables present in circuit analysis is indeed in the thousands to millions; hence the normality assumption is usually valid. However, the number of tasks usually present in application task graphs is at most a few hundred; hence the resulting makespan distributions are often not normal. In fact, the distributions need not even be continuous; we see this in the IPv4 packet forwarding example. Analysis of models with asymmetric and discrete random distributions requires a general approach. Such general analysis approaches are computationally more expensive than analytical approaches. The need to keep analysis simple at a time when computational resources were limited led to the use of beta and triangular distributions for project management. However, increasing computing resources has made more accurate analysis techniques such as Monte Carlo simulations feasible. Such general approaches are also seen in financial risk analysis, where the distributions are similarly complex [Ruppert, 2006][Lai and Xing, 2008].

4.5 Conclusions

In this chapter, we described extensions to the application and performance models used in compile-time scheduling in order to capture variations in application performance. The variations in application performance arise due to input data dependent executions and jitter in memory access times inside the application. For the sake of the accuracy of performance modeling, it is necessary for the variability in such applications to be captured. In this work, we proposed to capture the variations using statistical models for task dependencies as well as task execution and communication times. The importance of capturing these variations inside a scheduling approach is two fold: (1) it allows for more accurate estimations of the makespan of the scheduled application, and (2) optimization techniques can rely on these better estimates of performance to yield more optimal schedules. In order to achieve accurate estimates of application makespan, we proposed a Monte Carlo based statistical performance analysis procedure. We shall show over the next two chapters how this procedure is used inside statistical optimization algorithms for compile-time scheduling.

In this chapter, we studied the performance variations present in two applications: IPv4 packet forwarding and H.264 video decoding. These two applications show differing extents of variation in performance. An application like H.264 is highly control dominated. Execution times of individual macroblocks can vary by an order of magnitude depending on macroblock type and memory access times. It is of high importance to account for the variability when scheduling such applications. The variations in IPv4 forwarding is somewhat more limited in scope: only the lookup tasks of the forwarder exhibit variability in execution times. As such, it may not be as important to account for the variability in applications such as IPv4 forwarding. However, it is important to note that applications in many fields are becoming more complex and control-dominated. For instance, in the video codec space, we observe that the trend has been from simple and less efficient coding schemes like Motion-JPEG and MPEG-1 towards more efficient but logically complex and control-dominated coding schemes such as H.264. We expect similar trends in other application areas such as networking as standards such as IPv6 and video telephony come in. As such, we expect the importance of accounting for variations in applications to increase.

Chapter 5

Statistical Optimization

In Chapter 4, we motivated the need to account for the variations present in applications in the form of task execution times and dependencies during task allocation and scheduling. We captured the variations in applications using a statistical model. We then defined the optimization problem of obtaining a task allocation and schedule that minimizes a given percentile of the makespan distribution of the application. As a first step to solving this problem, we presented a Monte-Carlo based analysis procedure to obtain the makespan distribution (and the required percentile) given a valid task allocation and schedule. In this chapter, we use this procedure to develop optimization procedures to find the task allocation and schedule that minimizes a given percentile of the makespan distribution. In particular, we develop statistical analogues to the heuristic and simulated annealing optimization techniques used in the task allocation and scheduling problem for static models as described in Chapter 2.

5.1 Statistical task allocation and scheduling onto multiprocessors

We present a brief summary of the statistical task allocation and scheduling problem presented in Section 4.2.4. The inputs to the optimization problem are a task graph $G(V, E, L)$, the performance model (w, c) of the application and the architecture model (P, C). The nodes V of the graph represent the tasks in the application and the edges E represent the dependencies between tasks. L defines the probability of existence of each edge. The performance model parameters w and c represent the statistical distributions of the execution and communication times of the tasks. The architecture model is defined by P, the set of processors in the architecture, and C, the communication links between these processors. The output of the optimization is a valid allocation A that

maps tasks onto processors and a schedule S that orders tasks assigned to each processor. The allocation and schedule must respect the dependencies in the application (Section 4.3.1). The objective of the optimization is to obtain the best makespan for the application. However, in the presence of variations in task execution times and dependencies, the makespan is not a single number but is also variable. A useful optimization metric for soft real-time applications is to optimize for a fixed percentile η of the makespan distribution of the application. Such an optimization provides the best possible guarantee on the performance of the application while meeting a required Quality of Service (QoS) guarantee. For example, an IPv4 packet forwarding application may require that 99% of its inputs complete within a specified time frame. In this case, the makespan is best measured as the 99^{th} percentile of the makespan distribution. The fixed percentile η is an user-defined input to the optimization problem that is dependent on what QoS needs to be guaranteed for the application.

The optimal task allocation and schedule depends on the percentile of makespan that is being considered. For instance, in Section 4.3.4, we compared the makespans obtained from three different schedules (shown in Figures 4.12 (a) and (b), and Figure 4.16) for the IPv4 packet forwarding application mapped onto a pipeline of three processors, and showed the schedule in Figure 4.12(a) has the best 95^{th} percentile of makespan (among the three schedules), while the schedule in Figure 4.12(b) has the best 99^{th} percentile of makespan. Since the optimality of a schedule is influenced by the given percentile, any optimization procedure for the statistical task allocation and scheduling problem must take the percentile as an input and must be tuned for specific percentiles.

5.2 Techniques for Statistical Optimization

In this section, we will discuss two techniques for solving the statistical task allocation and scheduling problem. The first of these is a variant of the dynamic list scheduling algorithm which was described in Section 2.3.1. The second is a simulated annealing algorithm which can be adapted with very little change from the algorithm in Section 2.3.2. These two techniques offer different tradeoffs between the quality of the schedule and the time taken to compute the schedule.

The difference between the algorithms described in this chapter and those in Chapter 2 arise from the variations present in the statistical models. In the presence of variations, the conditions for the validity and optimality of schedules varies from those for a static model. Further, the makespan results must be customized to specific percentiles. We describe below how we adapt the heuristic and simulated annealing algorithms to meet these requirements.

5.2.1 Statistical Dynamic List Scheduling

Dynamic List Scheduling(DLS) is a popular heuristic algorithm that has been widely used in multiprocessor scheduling problems. The algorithm was initially developed for scheduling task graphs with static performance models onto fully connected multiprocessors [Sih and Lee, 1993]. We have shown in Chapter 2 that the DLS algorithm produces close to optimal solutions for static models and fully connected architectures. Since heuristics like DLS offer the promise of quickly obtaining close to optimal solutions (for certain architectural topologies), they form an important part of a toolbox of scheduling solutions.

In this section, we describe how to implement the DLS algorithm for statistical models. We first summarize the algorithm presented in Chapter 2 for static application models, and then highlight the changes that we need to make for statistical models.

In Section 2.3.1, we described a dynamic list scheduling heuristic for solving the task allocation and scheduling problem with static models. Algorithm 2.1 presented the structure of such an algorithm. The algorithm proceeds as a sequence of iterations. Each iteration involves allocating and scheduling a single task to a processor, while keeping in mind the scheduling decisions made in previous iterations. Recall that the key step in the dynamic scheduling algorithm is to define a priority metric (called the *dynamic level*) that decides the next task to be allocated and the processor to which it is allocated. This metric is recomputed at each iteration of the DLS algorithm of Algorithm 2.1 and is hence "dynamic".

The algorithm that defines the dynamic levels given a partial allocation A and schedule S was described in Algorithm 2.2. The dynamic level of a (task v, processor p) pair is computed as the difference between (1) the *static level* of the task (represented as $SL(v)$ in Algorithm 2.2) which is the longest path from the node representing task v to any other node in the task graph, and (2) the earliest start time of the task v on the processor p (represented as $ES(v, p, A, S)$). The (task, processor) pair with the highest dynamic level is chosen for allocation at each step. The static level $SL(v)$ of a task is computed by a breadth-first longest path computation on the task graph G. The earliest start time of a task on a processor $ES(v, p, A, S)$ is defined as the maximum of two parameters: $DA(v, p, A, S)$ which represents the earliest time that all predecessors of task v finish executing and communicating their results to v, and $TF(p, A, S)$ which represents the earliest time that processor p is free after executing all tasks assigned to it so far. Both DA and TF depend on the partial schedules computed so far.

Dynamic levels for statistical models

The key difference between the DLS algorithms for static and statistical models lies in the definition and computation of the dynamic level metric. In the presence of variability in task execution times and dependencies, both the static levels SL and the earliest start time of the task on a processor ES are statistically varying quantities. In order to compute these values, we can use our statistical analysis procedure described in Chapter 4. Recall that our statistical analysis procedure is based on a Monte Carlo simulation procedure. In Section 4.3.3, we described the key steps in the analysis. As a first step, we randomly pick a set of 1000 samples for all execution and communication times and dependencies in the task graph. In order to compute the static level, we must perform (for each sample) a standard linear-time longest path computation from each node in the task graph. We then obtain a set of 1000 values for the static level $SL(v)$ for each node v. We then aggregate all $SL(v)$ values obtained into a probability density function for each task. We can adopt a similar procedure for computing the earliest start time of task v on processor p. We perform this computation by adding a set of *ordering* edges representing the partial schedule A and S computed so far (Section 4.3.2). We can then directly compute the earliest start time of a task on a processor $ES(v, p, A, S) = \max\{DA(v, p, A, S), TF(p, A, S)\}$ as the longest path from any node in the graph to node v, taking care to add communication times between tasks that are allocated to different processors. This technique is analogous to the makespan computation procedure of Section 4.3.2, with the notable difference that the allocation A and schedule S constitutes an intermediate solution where not all tasks have been allocated or scheduled, and hence does not fully satisfy the constraints of the optimization problem.

Given the distributions of the static levels $SL(v)$ and earliest start times $ES(v, p, A, S)$, we can also compute the dynamic level, defined as the difference between these two values, as a distribution. However, the characterization of the dynamic level as a distribution is unsatisfactory as we need to be able to compare dynamic levels for different pairs of tasks and processors. Further, the comparisons between different dynamic levels must depend on the required percentile of makespan η, since the quality of a schedule depends on the percentile.

In order to achieve both these aims, it is most convenient to define the dynamic levels as a single number that is dependent on the value of η, the required percentile. We achieve this by defining the statistical dynamic level of a (task v, processor p) pair $SDL(v, p, A, S)$ as the difference between the η^{th} percentiles of $SL(v)$ and $ES(v, p, A, S)$. The η^{th} percentiles of $SL(v)$ and $ES(v, p, A, S)$ distributions are defined to be the statistical static level $SSL(v)$ and the statistical earliest start times

$SES(v, p, A, S)$ respectively.

$$SDL(v, p, A, S) = SSL(v) - SES(v, p, A, S)$$

where

$$SSL(v) \ = \ \eta^{th} \text{ percentile of } SL(v)$$
$$SES(v, p, A, S) \ = \ \eta^{th} \text{ percentile of } ES(v, p, A, S)$$

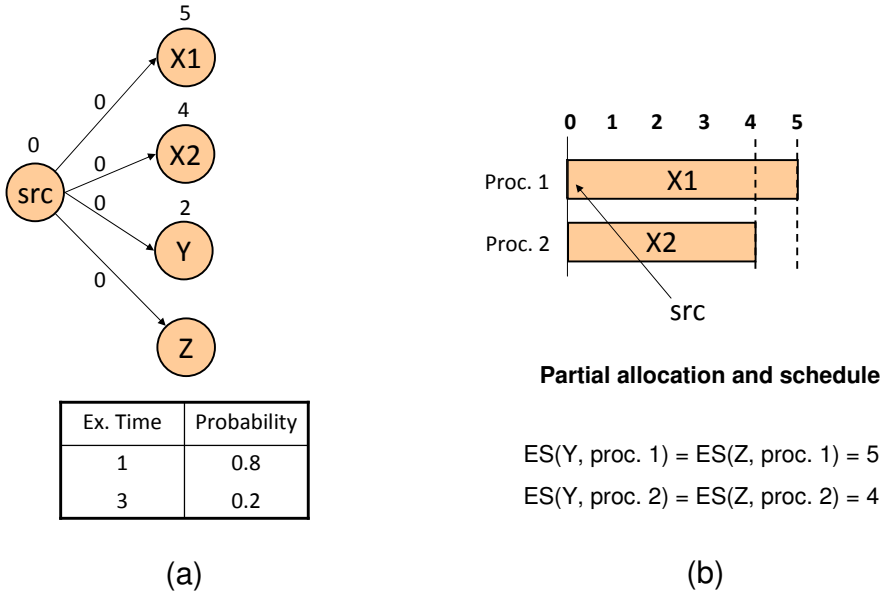


Figure 5.1: (a) An example task graph for statistical scheduling and (b) a partial allocation and schedule of tasks src, X1 and X2 onto two processors.

We now show by means of a simple example that the dynamic level and the scheduling choices made by the algorithm change with the value of the input makespan percentile η. The task graph shown in Figure 5.1(a) with 5 nodes needs to be scheduled onto two processors (processor 1 and processor 2). Nodes X1, X2 and Y have deterministic execution times of 5, 4 and 2 respectively, while node Z can have an execution time of either 1 (with 80% probability) or 3 (with 20% probability). The src node has an execution time of 0. For simplicity sake, let us set all communication times to zero. For this example, the static levels $SL(v)$ for nodes X1, X2, Y and Z have the same distributions as their respective execution times. The src task must be scheduled first, since this is the only task with no input dependencies. Tasks X1 and X2 are then scheduled as they have the

highest dynamic levels. Until this point in the algorithm, the tasks chosen do not exhibit variability in execution times, and the decisions made will be the same as the DLS algorithm in Chapter 2. Now let us consider the situation where we have finished scheduling the src node followed by nodes X1 and X2 as shown in Figure 5.1(b). We are trying to decide whether to schedule Y or Z next and which processor to schedule it on. For this example, the value of the earliest start time $ES(v, p, A, S) = 5$ for $p = 1$ and 4 for $p = 2$ (irrespective of whether x = Y or Z). It is clear that it is better to schedule the next task on processor 2. The choice is then as to which task should be scheduled on processor 2. The decision here is more complex as the static level of task Z could either be smaller or larger than the static level of task Y, depending on the percentile at which we sample the static level distribution of Z.

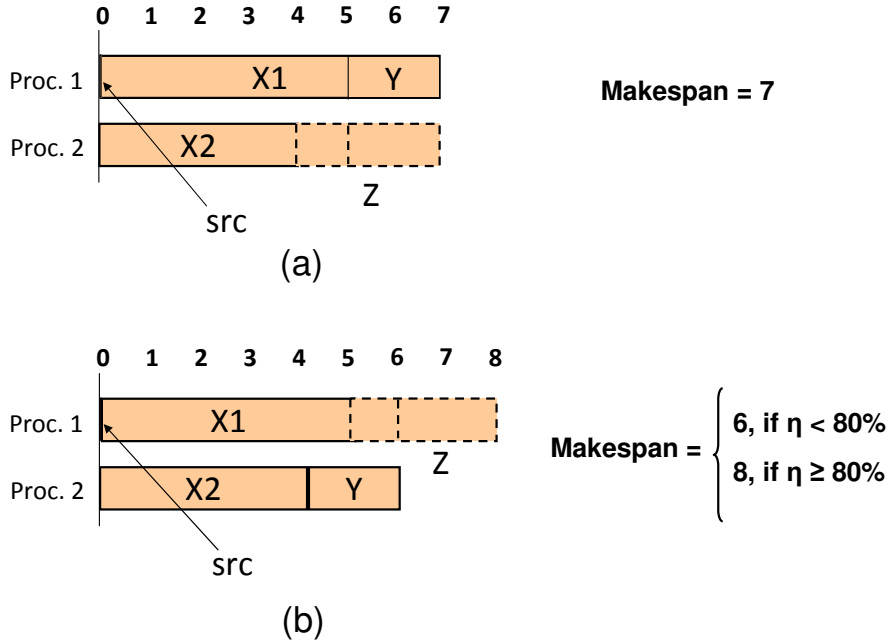


Figure 5.2: Two valid schedules for the task graph in Figure 5.1(a).

We now consider two cases depending on the value of the required percentile η. We first consider $\eta \geq$80%. If we sample the static level of Z at such a percentile, then task Z would have a higher static level ($SSL(Z) = 3$) than task Y ($SSL(Y) = 2$). The dynamic level for the task processor pair $(Z, 2)$ will then be the highest among all pairs, and the algorithm will pick task Z to be scheduled first on processor 2. Task Y would then be scheduled on processor 1. This schedule is shown in Figure 5.2(a). The decision to schedule Z first is the correct decision to minimize

makespan. The makespan of the schedule obtained above is a deterministic value of 7 irrespective of the value of η. The alternative decision of scheduling Y on processor 2 and Z on processor 1 (shown in Figure 5.2(b)) results in a makespan that can either take the value 6 (whenever task Z has an execution time of 1) or 8 (when task Z has an execution time of 3). At $\eta \geq 80\%$, this makespan takes the value 8, which is longer than the makespan of the schedule chosen by the statistical DLS algorithm. The other case is $\eta < 80\%$, in which case the schedule shown in Figure 5.2(b) has a makespan of 6 and is better than the schedule in Figure 5.2(a). We can see that the statistical DLS algorithm picks the right schedule in this case as well: $SSL(Y) = 2$ and $SSL(Z) = 1$ (the η^{th} percentile of $SL(Z)$ with $\eta < 80\%$), and hence the dynamic level of (Y,2) is greater than the dynamic level of (Z,2).

It is also possible that the earliest start time of a task on a processor $ES(v, p, A, S)$ is a statistically varying value. For instance, this occurs in the example in Figure 5.1 after we have scheduled all the tasks. Consider, for example, the schedule in Figure 5.2(b). While the earliest time that processor 2 becomes free is a deterministic value (6), the time that processor 1 becomes free is a distribution. If we wanted to schedule a new task on these processors, the choice of the processor to schedule it on will depend on the value of η. At $\eta < 80\%$, processor 1 has a finish time of 5, which is lower than the finish time of processor 2. On the other hand, processor 1 has a finish value of 8 when $\eta > 80\%$. Thus processor 1 would be picked for scheduling the next task if $\eta < 80\%$ and processor 2 would be picked otherwise. This is taken into account by the $SES(v, p, A, S)$ value as defined previously.

Algorithm pseudo-code

We now formally describe a list scheduling heuristic that uses the statistical dynamic levels described above for scheduling task graphs with statistical performance models onto multiprocessors.

The algorithm for statistical dynamic list scheduling is outlined in Algorithm 5.1. The inputs to the algorithm are the task graph $G = (V, E, L)$, the set of processors P, the performance model (w, c) and the required makespan percentile η. The output is the heuristic statistical makespan. Note that the processor topology C of the architecture model is not an input here: the heuristic assumes a fully-connected architecture. Such limitations on the scheduling problems are normally found in heuristic methods and are a key drawback of such methods.

The algorithm first initializes the allocation and schedule (line 1). It then computes the static

Algorithm 5.1 STATDLS(G, w, c, P, η) $\rightarrow$ *makespan*

// task allocation and start time variables
1 $A(v) = \epsilon,\ S(v) = \epsilon,\ \forall v \in V$
// obtain static level distributions by Monte Carlo simulations
2 $SL(v) =$ MCSIMULATE (G, w, c, P)
3 $SSL(v) = \eta^{th}$ percentile of $SL(v)$
4 **for** $(i = 1 \ldots |V|)$
// choose next task processor pair to schedule
5 $(v, p) =$ STATDLSDECIDE$(A, S, G, w, c, P, SSL, \eta)$
// update schedule based on selection
6 $S(v) = ES(v, p, A, S), \quad A(v) = p$
7 **return** η^{th} percentile of $\max_{v \in V} S(v) + w(v)$

level of tasks $SL(v)$ as the longest path in graph G using a Monte Carlo analysis technique (line 2). The statistical static level $SSL(v)$ is then computed by reading off the η^{th} percentile of $SL(v)$ (line 3). The algorithm then proceeds in $|V|$ steps (line 4). As in the DLS algorithm in Algorithm 2.1, the algorithm maintains the current allocation A and the start time S of tasks assigned so far. The algorithm then chooses a single task v to allocate and a single processor p to allocate it on (line 5). Then the start time of task v on processor p is set to $ES(v, p, A, S)$ which is defined as:

$$ES(v,p,A,S) = \begin{cases} \infty & : \text{if } A(v) \neq \epsilon \\ \infty & : \exists (v_1, v) \in E, A(v_1) = \epsilon \\ \max\limits_{\substack{v_1:(v_1,v)\in E \\ \vee A(v_1)=p}} S(v_1) + w(v_1) & \\ \qquad + c\left((v_1, v), (A(v_1), A(v))\right) & : \text{otherwise} \end{cases}$$

When computing $ES(v, p, A, S)$, we add the set of ordering edges (Section 4.3.2) $\{(u, v)\ :\ A(u) = p\}$ to graph G. These edges ensure that task v is scheduled after all previously allocated tasks on processor p (line 6). After adding these edges, $ES(v, p, A, S)$ can be computed as the maximum finish plus communication times of all predecessors of v (including those coming from dependence edges and ordering edges). The max operation is computed using a statistical Monte Carlo approach. After all tasks have been scheduled, the maximum finish time of any task in the graph yields the makespan distribution. The η^{th} percentile of this distribution is returned as the makespan (line 7).

Algorithm 5.2 shows the procedure used to decide the (task,processor) pair returned in line 5 of Algorithm 5.1. The algorithm computes a priority for every (task, processor) pair (line 1). The priority is computed as the difference between $SSL(v)$, the statistical static level of the task

Algorithm 5.2 STATDLSDECIDE($A, S, \eta, G, w, c, P, SSL, \eta$) $\rightarrow$ $(v \in V, p \in P)$

1 **foreach** $(v \in V, p \in P)$
  // compute statistical dynamic level for each task processor pair
2 $SDL(v,p) = SSL(v) - SES(v,p,A,S)$
  // return task processor pair with highest dynamic level
3 **return** $\arg\max_{v \in V, p \in P} DL(v,p)$

and $SES(v,p,A,S)$, the earliest start time of the task on the processor (line 2). The pair with the highest priority is then returned (line 3). The $SES(v,p,A,S)$ value is computed as the η^{th} percentile of the $ES(v,p,A,S)$ distribution that has been previously defined.

We now show that the resulting schedule is valid and satisfies the dependence and ordering constraints of Section 4.3.1. The dependence constraint enforces that the task dependencies are met probabilistically: a task can start only after all its predecessors complete. The ordering constraint enforces a total ordering of the set of all tasks assigned to a processor. There is no probabilistic nature to these inequalities, since they arise due to the hard constraint that tasks allocated to a single processor should not overlap.

At each iteration of Algorithm 5.1, we assign the start time of a task to the maximum of the finish plus communication times of all predecessors. Such predecessors include both the original dependence edges in the task graph (which can have a probability of existence of less than 1) as well as the ordering edges from the tasks previously assigned to the same processor (which will have probability of existence of 1). The computation of the maximum is performed statistically using a Monte Carlo analysis technique. This statistical max computation straightaway encodes both the constraints of Section 4.3.1: the dependence edges ensure that the dependence constraint is satisfied, while the ordering edges ensure a global order among tasks assigned to the same processor. The ordering edges are given a probability of 1.0 (Section 4.3.2), and hence the probabilistic max reduces to the deterministic equivalent for these edges. This is taken care of by the Monte Carlo analysis. For those edges that are both dependence and ordering edges, we replace the dependence edge probability with the ordering edge probability of 1.0 - hence ensuring strict non-overlap of dependent tasks within the same processor.

The statistical DLS algorithm outlined above reduces to the deterministic DLS algorithm of Algorithm 2.1 if there is no variation in the application. As in the definition of the deterministic Dynamic List Scheduling algorithm, the definition of static levels used changes if the system is heterogeneous or if additional constraints are introduced in the mapping. For heterogeneous sys-

tems, the average execution time of a task on all processors is generally used to compute the static levels [Sih and Lee, 1993]. In case task execution times vary, then the statistical average must be employed. Extensions to the priority metric to deal with irregular multiprocessor architectures have also been proposed to the DLS algorithm [Bambha and Bhattacharyya, 2002]; these can also be used in the statistical algorithm. However, such extensions to a heuristic algorithm like DLS are non-trivial to make and involve a significant amount of customization. It is necessary, in general, to develop techniques that can more flexibly handle additional restrictions to the scheduling problem such as topology restrictions, constraints on task grouping or clustering of tasks. This motivates the use of a generic algorithm such as Simulated Annealing to solve the statistical scheduling problem.

5.2.2 Simulated Annealing

While heuristics have been found effective for certain multiprocessor scheduling problems, they are often difficult to tune for specific problem instances. There are often additional constraints imposed on the problem in the form of topology constraints, constraints on task grouping and so on. A technique that has found success in exploring the complex solution space in scheduling problems has been Simulated Annealing. The simulated annealing technique is flexible and can accommodate a variety of objectives.

We have already discussed the use of Simulated Annealing as a generic and flexible combinatorial optimization approach in Chapters 2 and 3. We described the basic structure of a simulated annealing algorithm in Algorithm 2.3. In order to use simulated annealing to solve a particular optimization problem, it is necessary to tune the selection of the COST, MOVE, PROB and TEMP functions, and the initial and final temperature parameters t_0 and t_∞ (Section 2.3.2). In Section 2.3.2, we discussed the choice of these functions and parameters to solve the task allocation and scheduling problem for static models.

In the context of task allocation and scheduling with statistical models, a valid simulated annealing state corresponds to a valid allocation and ordering of tasks subject to the constraints in Section 4.3.1. The MOVE function defines the transitions between these states. As in Section 2.3.2, our transitions are based on moving a single task from one processor to a random location on another processor. After each transition, the COST function evaluates the makespan of the valid allocation and schedule corresponding to the state. For statistical optimization, the cost of a state is the makespan obtained by the statistical analysis procedure in Section 4.3. Note that this analysis procedure takes into account the required makespan percentile η. The decision to accept or reject

a transition is defined by the PROB and TEMP functions. We do not change the definitions of these functions from the ones described in Section 2.3.2 for static models.

The main drawback of using a Monte Carlo based statistical analysis technique for the COST evaluation is the high runtime involved in performing Monte Carlo iterations. A simulated annealing based optimization procedure to schedule a few hundred tasks can take more than an hour to complete if 1000 Monte Carlo iterations are used to evaluate each state. However, we can avoid performing the full Monte Carlo analysis for each state by exploiting the way in which the MOVE function picks the next state. We describe this in the next section.

Use of Incremental Timing Analysis to speed up cost evaluation

The Monte Carlo statistical analysis method is used inside the inner loop of the simulated annealing engine. Performing a full Monte Carlo analysis inside each iteration can become computationally challenging and make Simulated Annealing an unattractive choice as a scheduling technique for statistical models.

An incremental analysis is useful when we already have an analysis of a graph, and the graph structure is then perturbed a little. If edges are added or removed from the graph, then it is just the nodes in the fanout cone of the modified edges that need to be analyzed again. The start times (produced by statistical analysis) of all other nodes in the graph remain the same as before. If only a few edges are modified, this can lead to substantial savings in the number of nodes analyzed. Such incremental techniques have been used in the context of circuit simulation and physical synthesis to compute circuit delays. The limitation of the nodes to be considered for analysis has been called *level limiting analysis* in [Visweswariah *et al.*, 2006]. In the context of statistical scheduling, the analysis is a longest path computation on the task graph with additional ordering edges added to it. The longest path computation is usually performed using a breadth-first traversal of the graph. In the incremental analysis, we do the same - except that we only need to perform a breadth-first traversal of the graph starting from the target nodes of the modified edges.

We now study the change in the structure of the ordering edges as we perform a move inside the simulated annealing algorithm. The MOVE function in the simulated annealing algorithm only shifts a single task to a new processor. Therefore, the only change in the task graph structure is involved with the ordering edges of the task that is moved. Before the move, the task that is being moved had two ordering edges associated with it - one from its preceding task in the processor to which it was allocated and the other from its succeeding task in the same processor. Figure 5.3

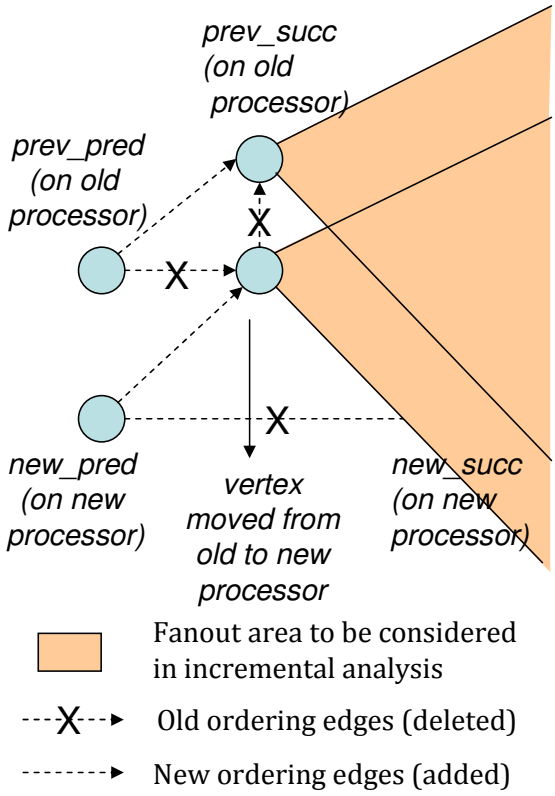


Figure 5.3: Incremental Statistical Analysis..

shows how these ordering edges change with a single transition. There are a total of six edges that change due to the transition. First, the edges from the predecessor and successor of the task in the old processor are removed. In order to maintain a global order of tasks in the old processor, an ordering edge is added from the predecessor to the successor of the task in the old processor. After a new processor and a new location for the task is found, the ordering edge from the new predecessor of the task and the edge to the new successor are added. Finally, the ordering edge between the predecessor and the successor of the task in the new processor can be removed.

The fanout of the six edges that are modified overlap significantly. In particular, the fanout of the successor task in the old schedule and the task that has been relocated are the only two tasks whose fanouts have to be considered. In Figure 5.3, the area to be considered in considered in incremental analysis has been highlighted in orange.

Another facet of incremental analysis is *dominance-limiting analysis*. If a particular modified edge does not affect the arrival time of the target node, then it has been dominated by the node's

other fan-in edges. Then we do not need to proceed with a breadth-first traversal of that edge. In the context of our Monte Carlo analysis, we perform a number of deterministic runs. If an edge has been dominated in any of these runs, then there is no need to carry forth the breadth-first analysis for that particular run. We could also decide not to carry forward the results of the entire node if all the runs are dominated.

The techniques of level limiting and dominance limiting analysis together can significantly reduce the number of operations to be considered in the course of Monte Carlo analysis. By adopting these techniques inside the Simulated Annealing algorithm, we can obtain a speedup of up to 5X over performing full Monte Carlo analysis. Such a speedup helps to keep simulated annealing a viable scheduling technique.

5.2.3 Deterministic Optimization Approaches

An alternative approach to the statistical optimization approaches described so far in this section is to use optimization techniques that rely on the deterministic scenario-based analysis techniques described in Section 4.3.3. Such a technique would use worst-case or common-case estimates of task executions and dependencies (as described in Section 4.3.3), thereby approximating the statistical application model with an easily analyzable static model. The deterministic analysis can then replace the statistical Monte Carlo analysis in the algorithms described in this chapter (which will yield the deterministic algorithms described in Chapter 2). It is to be noted that the common-case analysis must ensure that the ordering of tasks on processors do not violate any precedence edges, including the edges with a probability of existence below 0.5. In a DLS heuristic, this is done by ensuring that the earliest start time computation of a task on a processor returns ∞ unless all predecessors of the task in the original task graph are complete. In a SA based algorithm, the check for schedule validity must explicitly check the validity of task orderings.

However, such optimization approaches may still not yield allocations and schedules that are valid under the constraints of Section 4.3.1. In particular, the start times for all tasks obtained through deterministic optimization will naturally be approximate constants and not distributions. The exact value of the makespan returned by these algorithms will likewise be inaccurate. One viable workaround to this problem would be to only use the task allocation and ordering of tasks returned by deterministic optimization, and ignore the deterministic values for task start times. Such a policy is in fact widely used in implementing schedules, and is popularly called self-timed scheduling [Poplavko *et al.*, 2003] [Moreira and Bekooij, 2007]. Such a system is typically implemented

using synchronization primitives to regulate task initiations and terminations. One specific method is to use a logical queue between tasks, with tasks being triggered when inputs to the task arrive from its predecessors. Under the assumption of self-timed scheduling, schedules obtained by deterministic optimization can also be used in a statistical context. The makespan obtained through a deterministic self-timed schedule will vary depending on the inputs and architectural parameters. This variation can be computed by using the statistical Monte Carlo analysis procedure of Section 4.3.3. The required makespan percentile η of the resulting makespan distribution is the result of the optimization.

Although the deterministic scheme proposed above does factor in the variations in the application model, it only does so after all optimization is done. The evaluation and comparisons of different schedules is performed using static analysis. In Section 4.3.4, we showed that static analysis can compare different valid schedules incorrectly. An optimization scheme based on such inaccurate comparisons leads to the selection of suboptimal schedules. We will quantify the extent of this sub-optimality in Section 5.4.

5.3 Related Work

In the previous chapter, we outlined works that develop statistical models in the context of both task scheduling [van Gemund, 1996] [Gautama and van Gemund, 2000] as well as other areas such as circuit timing analysis [Visweswariah *et al.*, 2006] [Orshansky and Keutzer, 2002] [Najm and Menezes, 2004]. Such works also develop analysis and optimization techniques that utilize these models. In the scheduling context, algorithms that use statistical task execution models have been proposed for real-time scheduling of periodic tasks with deadlines onto multiprocessors [Manolache *et al.*, 2004] [Manolache *et al.*, 2007]. Such methods assume that tasks are periodic and have individual deadlines. In this work, we focus on a different optimization problem of scheduling aperiodic tasks to optimize for schedule length.

Statistical timing analysis has been well studied recently for circuit timing in the presence of process variations. As we mentioned in the previous chapter, the models used in this context are primarily normal (or near-normal) distributions that allow for fast analysis of large graph with millions of nodes [Visweswariah *et al.*, 2006] [Orshansky and Keutzer, 2002]. Such models and analysis have been used in varied optimization problems involving power optimization [Srivastava *et al.*, 2004] [Mani *et al.*, 2005] and gate sizing [Mani and Orshansky, 2004]. The problems are usually modeled as convex optimization problems. Scheduling, on the other hand, is an inherently

discrete and non-convex optimization problem. Moreover, most scheduling problems involve only a few hundred tasks and hence we can afford to use optimization techniques that utilize a more expensive and general Monte Carlo analysis technique.

Optimizing compilers have used profiling-based techniques to solve the instruction scheduling problem [Chen *et al.*, 1994], but these have traditionally been based on average-case analysis. We target applications where high statistical guarantees on performance are required; and hence average-case scheduling is not directly useful.

The closest work to the techniques described in this chapter has come from the operations research community that has worked on using statistical optimization for job-shop scheduling problems [Beck and Wilson, 2007]. As in our current work, the work in [Beck and Wilson, 2007] tries to find a schedule to optimize for a fixed percentile of the finish time. However, it solves only the job-shop scheduling problem, which is a special case of the multiprocessor scheduling problem with the restriction that the tasks must consist of independent chains [Coffman, 1976]. The technique also uses only the means and standard deviations of task executions. In this work, we consider arbitrary distributions of task execution times.

5.4 Results

In this chapter, we have described optimization techniques for the task allocation and scheduling problem using statistical application models. In this section, we compare the makespans obtained by deterministic techniques with those obtained from the statistical optimization techniques described so far in this chapter.

5.4.1 Benchmarks

We use two sets of benchmarks. The first set consists of two practical applications - the IPv4 packet forwarding application and the H.264 video decoding application. The performance profile of the IPv4 application was collected on soft MicroBlaze cores. The task graph for the IPv4 application was unrolled 1-10 times to represent multiple input channels in the forwarder. The soft multiprocessor architectures shown in Figures 2.10 were used in our benchmarks. The H.264 application profile was collected on an 2.6 GHz Intel Core2 Duo machine. We used three different input video streams - the akiyo.264, foreman.264 and the manitu.264 video streams. For each stream, we collected execution time statistics for I-, P- and P-skip macroblocks, as well as the percentage

Name	Resolution	Avg. Prob. that a macroblock is of type		
		I	P	P-skip
akiyo	176 × 144	0.01	0.14	0.85
foreman	176 × 144	0.01	0.76	0.23
manitu	320 × 144	0.12	0.52	0.36

Name	Execution time variations (microseconds)								
	I			P			P-skip		
	BC	WC	CC	BC	WC	CC	BC	WC	CC
akiyo	14	44	30	14	48	16	3	22	4
foreman	14	52	26	14	72	28	3	26	4
manitu	12	70	28	14	82	26	3	29	4

Table 5.1: Salient characteristics of the H.264 video decoding algorithm for different input streams.

of time that a given macroblock belongs to each of these categories. A summary of the key characteristics of these streams are shown in Table 5.1. The table presents the average probability that a macroblock is an I-, P- or P-skip macroblock, as well as the execution times (best case, worst case and common case) of macroblocks of each of these types. Recall from Section 4.5 that we consider the H.264 application to be representative of future applications with respect to the extent of variability present in the application. For each input stream, we map the H.264 application on architecture models comprising of 4,8, 10, 12 and 16 processors. For each model, the processors were divided into two sets, communication between which was a factor of 4 more expensive than within the set (this models locality of processors on an on-chip multiprocessor network).

The second set of benchmarks consisted of a set of random task graphs obtained from Davidović et al. [Davidović and Crainic, 2006]. These cover a wide range of task graph structures and contain both easy and hard scheduling instances. For each benchmark, we assume that the execution times of tasks are normally distributed. The mean execution times were taken from the benchmarks. The standard deviations were chosen randomly in the range [0 - 0.7*mean] such that the average ratio of standard deviation to the mean of each task is 0.35. We chose a value of 0.35 as this was the average ratio for the two practical applications.

5.4.2 Comparison to deterministic scheduling techniques

In this section, we compare the makespans obtained by the deterministic worst-based and common-case techniques (Section 5.2.3) with the result of the statistical simulated annealing technique described in Section 5.2.2 at different makespan percentiles. The deterministic techniques also

use a simulated annealing optimization technique as described in Chapter 2 with the same annealing parameters as the statistical algorithm for a fair evaluation. A final Monte Carlo run is performed on the schedule returned by the deterministic optimization techniques in order to incorporate the makespan percentile η into the deterministic algorithms.

# Tasks	Arch. (a)						Arch. (b)					
	$\eta = 99\%$			$\eta = 95\%$			$\eta = 99\%$			$\eta = 95\%$		
	Det. WC	Det. CC	Stat. SA	Det. WC	Det. CC	Stat. SA	Det. WC	Det. CC	Stat. SA	Det. WC	Det. CC	Stat. SA
15	150	165	145	150	125	125	150	165	145	150	125	125
28	220	215	205	220	195	195	150	165	145	150	145	135
41	300	290	265	300	280	255	155	170	145	155	155	145
54	385	345	315	385	330	325	220	230	195	220	220	180
67	455	430	390	455	410	385	225	240	195	225	225	190
80	535	490	455	535	470	440	235	250	200	230	235	200
93	640	575	515	640	555	510	305	305	270	305	295	250
106	750	675	600	750	665	590	320	310	275	320	300	265
119	840	790	690	840	750	680	335	340	290	320	310	275
132	930	840	725	930	800	710	390	380	335	390	360	320
Avg. % degradation from Stat. SA	19.6	10.7	-	21.6	7.3	-	12.1	16.0	-	17.3	12.8	-

# Tasks	Arch. (c)					
	$\eta = 99\%$			$\eta = 95\%$		
	Det. WC	Det. CC	Stat. SA	Det. WC	Det. CC	Stat. SA
15	150	165	145	150	125	125
28	150	170	145	140	155	140
41	175	170	150	170	165	145
54	190	180	165	190	175	145
67	215	240	195	215	220	200
80	245	255	220	245	240	210
93	250	265	230	250	250	215
106	275	280	245	275	265	235
119	320	315	275	320	300	270
132	365	345	305	365	325	295
Avg. % degradation from Stat. SA	11.7	15.1	-	16.8	14.0	-

Table 5.2: Makespan results for the deterministic worst-case, deterministic common-case and statistical SA methods on task graphs derived from IPv4 packet forwarding scheduled on the architectures (a) and (c) of Figure 2.10.

Table 5.2 reports the results of the deterministic and statistical optimization techniques for the

IPv4 packet forwarding application on the three soft multiprocessor architectures of Figure 2.10. The first column shows the number of tasks in the application on unrolling the task graph from one to ten times. The other columns show the results of the deterministic worst-case (Det. WC), deterministic common-case (Det. CC) and statistical simulated annealing (Stat. SA) techniques at percentiles of 99% and 95% for the three different architectures. Table 5.3 shows the same comparison for the H.264 video decoding application. We use three different video streams. Column 2 shows the number of macroblocks (tasks) in each video stream. Column 3 shows the number of processors in the architecture. The remaining columns show the results of the deterministic and statistical approaches at η = 99%, 95% and 90%.

Bench-mark	# Tasks	# Procs.	η = 99%			η = 95%			η = 90%		
			Det. WC	Det. CC	Stat. SA	Det. WC	Det. CC	Stat. SA	Det. WC	Det. CC	Stat. SA
akiyo	101	4	289	295	273	260	275	249	250	242	237
		8	240	215	173	220	187	151	208	157	144
		10	175	164	151	155	141	131	147	133	123
		12	170	165	137	142	142	118	132	132	109
		16	153	165	121	136	140	103	126	97	94
Avg. % degradation from Stat. SA			22.2	19.5	-	24.2	19.6	-	24.9	8.7	-
foreman	101	4	860	903	791	806	780	760	791	760	740
		8	516	634	448	478	494	416	460	464	406
		10	467	510	366	462	473	344	430	381	328
		12	373	404	325	360	387	303	357	311	284
		16	336	367	270	304	307	242	295	263	225
Avg. % degradation from Stat. SA			18.1	31.1	-	20.7	23.5	-	21.6	11.9	-
manitu	182	4	1410	1435	1393	1343	1362	1333	1359	1285	1271
		8	984	1019	819	928	940	762	891	802	701
		10	801	796	695	732	735	649	745	660	598
		12	697	694	595	644	614	528	616	528	474
		16	638	704	527	572	560	461	564	446	406
Avg. % degradation from Stat. SA			16.9	19.8	-	19.7	18.8	-	30.1	10.7	-

Table 5.3: Makespan results for the deterministic worst-case, deterministic common-case and statistical SA methods on task graphs derived from H.264 video decoding application scheduled on the architectures with 4,8,10,12 and 16 processors.

Table 5.4 shows the results of the scheduling approaches on the benchmark with randomly generated task graphs. The graphs are classified by the number of tasks and edge density (the percentage ratio of the number of edges in the task graph to the maximum possible number of edges). Columns 3 through 8 report the percentage degradation of the deterministic worst-case and

# Tasks	Edge Density	Percentage degradation from Stat. SA					
		η = 99%		η = 95%		η = 90%	
		Det. WC	Det. CC	Det. WC	Det. CC	Det. WC	Det. CC
52	10	14.1	15.2	26.9	11.1	33.4	7.4
	30	16.2	26.6	24.7	20.8	24.4	19.5
	50	14.3	31.3	26.1	28.9	32.7	23.1
	70	12.4	22.3	30.9	18.4	37.2	13.7
	90	16.4	14.2	20.1	16.5	26.3	14.4
Avg. % degradation from Stat. SA		14.7	21.9	25.7	19.1	30.8	15.6
102	10	20.5	26.4	29.7	16.3	34.1	10.2
	30	14.1	24.4	26.9	21.8	33.4	16.4
	50	13.1	33.2	19.9	29.5	27.9	19.3
	70	16.4	21.4	26.9	14.9	33.4	16.1
	90	24.7	14.9	34.1	9.3	41.8	6.4
Avg. % degradation from Stat. SA		17.3	24.1	27.3	18.4	34.1	13.6

Table 5.4: Average percentage degradation of the worst and common-case deterministic schedules from the statistical SA schedule at different percentiles for random task graphs scheduled on 4,6 and 8 processors.

common-case makespans from the statistical makespan. The comparisons are done for makespan percentiles of 99%, 95% and 90%. Each row in the table is an average over the percentages for 4,6 and 8 processors. As before, the deterministic worst-case and common-case makespans are obtained through a final Monte Carlo run on the schedule returned by the respective deterministic algorithms.

Tables 5.2 and 5.3 show that statistical scheduling consistently performs better than both the deterministic techniques for both applications. In the IPv4 application, the deterministic worst-case makespan is anywhere from 11% to 22% off of the statistical makespan. For the H.264 application, this difference can go up to over 30%. The inefficiency of deterministic worst-case schedules is further corroborated from Table 5.4, with differences from statistical schedules going as high as 34%. This is because the worst-case scheduling incorrectly compares different schedules and hence yields a less optimal schedule than the statistical schedule. The improvement shown by the statistical algorithm over the worst-case schedule increases at lower η values. This is an expected trend as the worst-case makespan estimates become worse approximations of the true makespan at lower percentiles. The statistical technique, on the other hand, performs a Monte-Carlo analysis to find the true makespan. The deterministic common-case optimization is also inferior to the statistical technique. In contrast to the worst-case optimization, we can see from Tables 5.2 and 5.3 that

the common-case technique improves at lower percentiles. This is also expected: the common case makespans for the IPv4 and H.264 examples lie close to the 50^{th} percentile of the respective makespan distributions. As we decrease the required percentile, these become better approximations of the actual makespan. This trend is also seen in the results for the random benchmarks in Table 5.4.

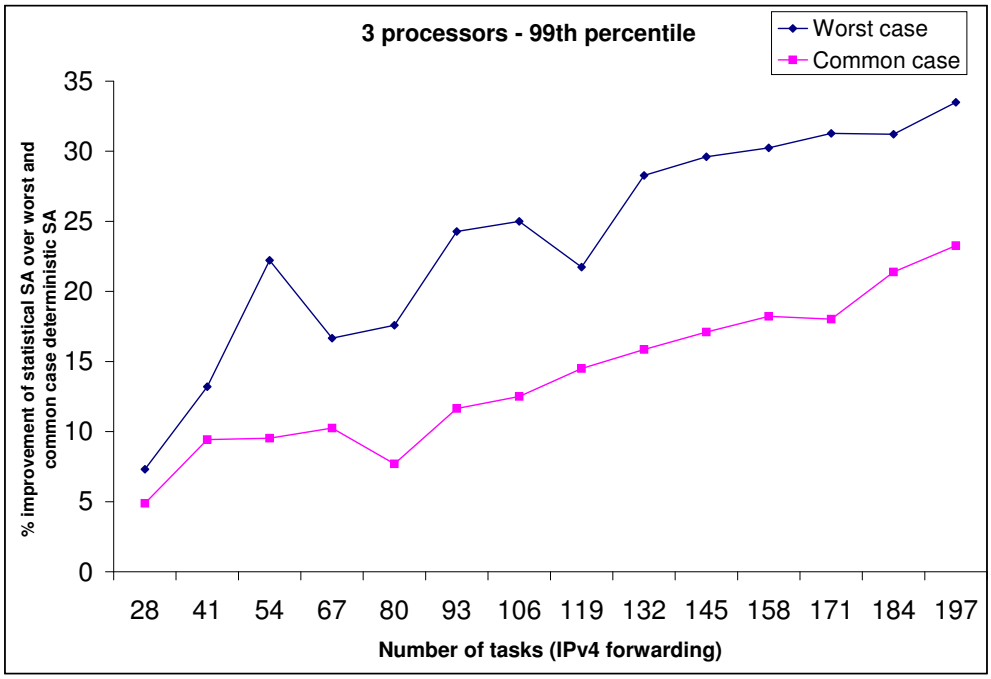


Figure 5.4: Comparison of statistical SA to deterministic worst-case and common-case optimization for different numbers of tasks in IPv4 packet forwarding.

Another interesting trend is that the percentage difference in makespans between the statistical and deterministic techniques increases with increasing problem size. Figure 5.4 shows the percentage improvements of the statistical versus the deterministic algorithms for the IPv4 forwarding application. We vary the number of tasks from 28 to 197 by unrolling the task graph up to 15 times. All runs were performed using $\eta = 99\%$. From the figure, we can see that the improvements of the statistical versus common-case schedules start out at about 5% for small task graphs and increase to about 25% for larger graphs. The percentage improvement is even higher (up to 35%) when compared to the worst-case schedules for large graphs. This trend occurs because large graphs tend to have more paths with different path lengths and distributions. The deterministic techniques have a higher chance of comparing the criticality of different paths incorrectly, leading to sub-optimal results. The statistical technique uses a Monte-Carlo technique to accurately estimate makespan, and hence does not suffer from this problem.

The improvement in makespan that we achieve with statistical scheduling comes at the expense of a significant increase in optimization time. We now need to perform a thousand Monte-Carlo iterations for each schedule that is evaluated over the course of the optimization. This can result in the SA technique taking up to two hours on large task graphs. The use of the incremental approach described in Section 5.2.2 brings down this runtime to a maximum of about 20 minutes for task graphs of size up to 200.

5.4.3 Comparison of statistical DLS and SA optimization techniques

We now compare the results of statistical DLS optimization to the results of the simulated annealing algorithm. We use task graphs obtained from the IPv4 packet forwarding and H.264 video decoding algorithms as our benchmarks. The architecture models for the IPv4 application capture the restrictions imposed by the soft multiprocessor networks shown in Figure 2.10, and the models for the H.264 application capture the effect of locality of processors on inter-processor communication times (as described in Section 5.4.1). These form additional constraints to the basic scheduling problem.

# Tasks	Arch. (a)				Arch. (b)			
	$\eta = 99\%$		$\eta = 95\%$		$\eta = 99\%$		$\eta = 95\%$	
	Stat. DLS	Stat. SA	Stat. DLS	Stat. SA	Stat. DLS	Stat. SA	Stat. DLS	Stat. SA
15	185	145	170	125	185	145	170	125
28	220	205	220	195	185	145	170	135
41	310	265	290	255	185	145	180	145
54	390	315	380	325	220	195	220	180
67	415	390	410	385	220	195	220	190
80	510	455	510	440	225	200	220	200
93	565	515	560	510	310	270	290	250
106	665	600	710	590	310	275	295	265
119	755	690	740	680	310	290	295	275
132	800	725	780	710	390	335	380	320
Avg. % improvement over Stat. DLS	-	13.4	-	15.1	-	17.2	-	18.7

Table 5.5: Makespan results for the statistical DLS and statistical SA methods on task graphs derived from IPv4 packet forwarding scheduled on the multiprocessor architectures of Figure 2.10(a) and (b).

Table 5.5 shows the makespan results of the DLS and SA algorithms on the IPv4 forwarding application on the multiprocessor architectures of Figure 2.10(a) and (b). Column 1 shows the number of tasks obtained from the task graph replication. Columns 2 to 9 show the makespan

obtained from the statistical DLS heuristic and statistical SA for the two architectures for makespan percentiles = 99% and 95%. We can see, the statistical SA algorithm consistently performs better than the heuristic. This is natural since the SA algorithm explores a larger portion of the design space. This does, however come at the expense of runtime. The statistical SA procedure can take up to 20 minutes, while all heuristic runs complete in under 3 minutes.

Bench-mark	# Tasks	# Procs.	η = 99%		η = 95%		η = 90%	
			Stat. DLS	Stat. SA	Stat. DLS	Stat. SA	Stat. DLS	Stat. SA
akiyo	101	4	281	273	261	249	237	237
		8	176	173	151	151	147	144
		16	125	121	110	103	94	94
Avg. % improvement over Stat. DLS			-	2.7	-	3.9	-	0.7
foreman	101	4	806	791	768	760	743	740
		8	466	448	428	416	426	406
		16	286	270	244	242	234	225
Avg. % improvement over Stat. DLS			-	4.0	-	1.6	-	3.1
manitu	182	4	1490	1393	1408	1333	1347	1271
		8	885	819	800	762	729	701
		16	559	527	498	461	434	406
Avg. % improvement over Stat. DLS			-	7.0	-	6.2	-	5.6

Table 5.6: Makespan results for the statistical DLS and statistical SA methods on task graphs derived from H.264 video decoding application scheduled on architectures with 4,8 and 16 processors.

However, the statistical SA algorithm does not improve the DLS result significantly for the H.264 example. Table 5.6 shows the results for the DLS and SA examples on 4, 8 and 16 processors for the three video streams of Table 5.1. We note that the difference between the DLS and SA results is always within 8%, and can be as low as less than 1%. This trend occurs because of the sparsity of dependence edges in the task graphs used in H.264 decoding. For task graphs with sparse dependencies, the scheduling problem simplifies to essentially a load balancing problem. The solution space for the load balancing problem is much smoother than the full scheduling problem, and as such the problem is easier to solve. In particular, list-scheduling heuristics are known to do very well on load-balancing problems. This is reflected in our results.

The number of dependencies in the task graph for H.264 decoding depends on the input video stream. In particular, the percentage of all macroblocks in the video stream that are I-macroblocks directly the density of dependence edges. Most video streams, including the three that we listed in Table 5.1 have a low percentage of I-macroblocks. This is primarily to increase coding efficiency:

P-macroblocks can be encoded using a smaller number of bits than I-macroblocks. Therefore, we can expect any H.264 task graph to have a sparse set of dependencies, and hence to be easy to schedule.

The choice of optimization method can thus be seen to depend on the nature of the application as well as the trade-off required between the quality of solution and runtime. Applications whose task graphs do not have many dependencies will not benefit much from using optimization schemes that explore significant portions of the design space over a simple list scheduling heuristic. Heuristics also work well for task graphs that are almost fully connected, since such graphs have a very small solution space of valid orderings and are hence easier to solve. However, a significant number of applications have task graphs with an intermediate number of edges. Such task graphs give rise to the most difficult scheduling instances and usually require a systematic exploration of the design space. For such graphs, the choice whether to use a heuristic algorithm or a more generic algorithm such as simulated annealing depends on the required tradeoff between computational speed, quality of solution and flexibility of the technique. In case there are additional constraints on the scheduling problem (including processor topologies, task clustering and heterogeneous inter-processor communication times), heuristics need to be modified to capture these constraints. Such modifications detract from the performance of heuristic techniques. Flexible approaches such as simulated annealing can consistently deliver better performance than heuristic techniques in the presence of additional constraints.

5.4.4 Summary

In this section, we first compared the makespans obtained by deterministic optimization techniques with those obtained from statistical optimization. We showed that the statistical simulated annealing algorithm performs consistently better than its deterministic worst-case and common-case counterparts for two realistic applications: IPv4 packet forwarding and H.264 video decoding. The difference in the makespan obtained by deterministic and statistical techniques can be as much as 20% for the IPv4 forwarder and up to 30% for the H.264 video decoder. The percentage improvement further seems to increase for larger task graphs. This motivates the need to consider statistical models and optimization methods.

We also compared two different statistical optimization methods: a statistical list scheduling heuristic and a simulated annealing based algorithm. We showed that the simulated annealing algorithm achieves makespans that are 15-20% better than the list scheduling heuristic for the IPv4

packet forwarding algorithm. The trade-off is in the time required for optimization: a simulated annealing algorithm can take up to 20 minutes to execute while all heuristic runs complete in under 3 minutes. We also noted that for the H.264 video application, there was no significant difference in the makespan produced by the heuristic and the simulated annealing algorithms. This is primarily because the task graph for the H.264 applications is very sparse and has few dependencies. For task graphs with sparse dependencies, the scheduling problem is easy to solve. In particular, list scheduling heuristics perform very well on such graphs.

Chapter 6

Constraint Optimization approaches to Statistical Scheduling

In this chapter, we revisit our decomposition-based constraint optimization approach proposed in Section 2.3.3 in the context of the statistical models proposed in Chapter 4. A constraint-programming based optimization technique offers the advantage of maintaining a high degree of flexibility in the range of additional constraints that can be added to the scheduling problem. They also offer the promise of obtaining exact solutions to an optimization problem when given enough computation time. Even when optimal solutions cannot be obtained in reasonable time-frames, these techniques can offer bounds on the optimality of the solution obtained. These advantages motivate this study of constraint-programming based approaches to the statistical scheduling problem.

The main disadvantage of constraint-optimization approaches has been the fact that they do not scale well beyond very small scale problems of 30-50 tasks with approximately a thousand variables and constraints. A technique that has been shown to work for solving the static scheduling problem is to decompose the entire scheduling problem into a master and sub problem that are then solved in a series of iterations with a feedback between successive iterations (Chapter 2). Such a technique has enabled the use of constraint optimization techniques for medium scale scheduling problems up to 200 tasks with a few tens of thousands of variables and constraints. This motivates the use of decomposition-based approaches as good starting points to develop scalable constraint-optimization techniques for scheduling problems.

In this chapter, we first present a straightforward extension to the decomposition-based technique described in Chapter 2 in the context of statistical models. The master problem in the decom-

position decides an allocation and schedule, and the sub-problem performs a statistical analysis on that allocation and schedule. As before, these two problems are executed iteratively with a feedback loop between successive iterations. We then describe techniques based on pruning techniques and heuristic guidance of the search process that we can use to speed up this constraint optimization approach.

6.1 Decomposition based Approaches

In Chapter 2, we described a decomposition based constraint optimization approach to task allocation and scheduling with static models. The approach consisted of breaking up the problem into a "master" problem that finds an allocation and schedule and a graph-theoretical "sub" problem that analyzes the allocation and schedule to check for validity, and if valid, obtain a makespan. The two problems are executed iteratively: the result of the sub-problem is used to prune out parts of the solution space containing solutions that are either infeasible or are inferior to the best solution found so far. These are encoded as additional constraints to the master problem that is used during the next iteration. The efficiency of the technique is primarily dependent on the number of iterations required. This, in turn, is determined by how much of the solution space can be pruned by a single sub-problem iteration. In Section 2.3.3, we used two types of constraints: cycle-based and path-based constraints that help to prune the solution space efficiently. The flow of the decomposition algorithm was presented in Algorithm 2.4.

In the statistical context, the variables and constraints used in the master problem are unchanged from Section 2.3.3. In particular, we have three sets of Boolean variables: x_a, x_c and x_d encoding the allocation of tasks to processors, communication between tasks and ordering of tasks allocated to the same processor respectively. Constraints are added to ensure that the allocation is valid and that there is a global ordering of tasks allocated to the same processor. The communication variables between dependent tasks that are allocated to different processors is set so as to account for the communication time.

The main difference between the static and statistical models lies in the nature of the analysis in the sub problem. We first recapitulate the key steps in the sub-problem (presented in Section 2.3.3). The sub-problem must first check if the master problem solution is valid by checking for cycles in the scheduled graph. If an allocation and schedule returned by the master problem is found to be valid, the sub-problem has to perform two functions: (1) compute the makespan of the valid schedule, updating the best makespan found by the algorithm so far if necessary, and (2) find one

or more paths in the task graph (after adding ordering edges) that have a path length greater than or equal to the current best makespan. The key idea here is that the length of a path is a lower bound on the makespan of any schedule containing the path. Hence the presence of paths with a path length greater than or equal to the best makespan is in itself a sufficient reason to reject all schedules containing the path from consideration. Such paths are eliminated using path constraints that are taken into account in the next iteration of the master problem.

In a statistical context, we can use the statistical analysis procedure of Section 4.3.3 (instead of the static longest path analysis in Chapter 2) to find the makespan of a valid allocation and schedule. This procedure returns a distribution of makespans. Recall from Chapter 4 that the optimization objective in statistical scheduling is to minimize a given percentile η of the makespan distribution. The last step in the analysis is therefore to read off this percentile η of the makespan distribution to yield a single makespan number. This number is then used in the sub-problem in a similar fashion to the way the static makespan is used in Chapter 2. In particular, the makespan value is used to update the best makespan found so far. After updating the makespan, the sub-problem requires us to find one or more paths that have a path length at least as long as the updated best makespan m_{best}. In a statistical context, the lengths of paths are variable. The analogue to the path constraint in static scheduling is to find paths whose η^{th} percentile of path length is greater than the current best makespan. As in the static approach, this value is a lower bound to the makespan of any schedule that contains the path. Therefore, if we find a path whose η^{th} percentile is greater than the best makespan, it is valid to eliminate all schedules containing that path via a path constraint.

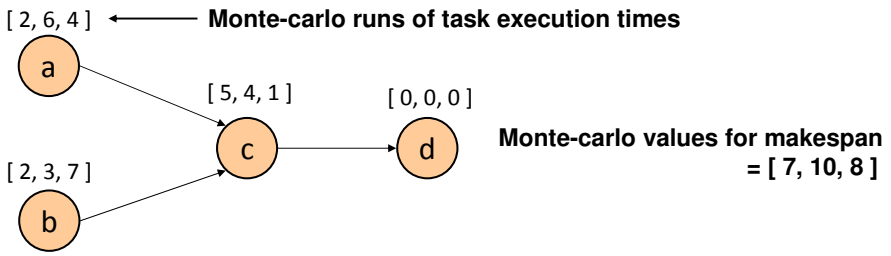


Figure 6.1: A task graph where no single path length determines the makespan.

However, there is a key difference between the path constraints for static and statistical scheduling. For static scheduling, we cannot fail to find a path that is longer than the current best makespan. This is because such a situation can only arise when the makespan of the current schedule is strictly smaller than the current best makespan, which is not possible as the current best makespan would

have been updated in that case. However, this useful property does not hold for statistical scheduling. In a statistical context, it is possible that there is no single path that is longer than the current best statistical makespan m_{best}, but still the overall current makespan is larger than m_{best}. As an example, consider the task graph in Figure 6.1. The graph consists of four nodes and has two paths that could potentially have a length greater than any given makespan. The task graph is annotated with the Monte Carlo values for the execution times of tasks, denoted by $[v_1, v_2, \ldots v_n]$. In the example, we have three Monte Carlo runs. The result of the makespan analysis procedure of Section 4.3.3 is also shown. Let us assume that the required percentile of the makespan is the 70^{th} percentile, which would be 8 in this example. Let us further assume that this was the best makespan found so far, and thus $m_{best} = 8$. In this scenario, we are required to find paths that have a 70^{th} percentile of path length greater than or equal to 8. For path $a \rightarrow c \rightarrow d$, the path length is given by $[7, 10, 5]$, which has a 70^{th} percentile of 7. The only other path $b \rightarrow c \rightarrow d$ has a path length of $[7, 7, 8]$, which also has a 70^{th} percentile of 7. Thus in this example, it would be incorrect to prune out schedules containing either path, since such schedules are only guaranteed to have a makespan greater than 7. We can only prune out schedules that have *both* paths. In other words, the two paths $a \rightarrow c \rightarrow d$ and $b \rightarrow c \rightarrow d$ are together responsible for the 70^{th} percentile of the makespan being larger than 8. Such a situation cannot arise in a static model, where there always exist one or more "longest" paths that determine the makespans by themselves.

Since no one path may determine the makespan in a statistical context, the path constraints added to the master problem in statistical scheduling needs to be modified to encode and prune a combination of paths. We do this by constructing a list of edges that is the union of the edges present in each of these paths. For the above example, we would construct the list $E_p = \{a \rightarrow c, b \rightarrow c, c \rightarrow d\}$. We then follow the same procedure as in Section 2.3.3: we construct a conjunction of the variables x_d and x_c corresponding to the ordering and communication edges among this list and eliminate the possibility of this conjunction of variables being present in the optimal solution.

In case there is more than one path that is responsible for the makespan, the constraint added to the master problem only eliminates the union of all these paths from consideration in future schedules explored. This would necessarily prune only a subset of the solution space that would be pruned by eliminating any one of these paths. Therefore, if there exists a single path (or a small set of paths) that determine the makespan of the schedule, it is important to find such paths so that a larger solution space can be pruned. As an example, in Figure 6.1, if the current best makespan is 7 instead of 8, then either path would individually have a path length equal to the best makespan. In such a case, adding the path $a \rightarrow c \rightarrow d$ (or $b \rightarrow c \rightarrow d$) as a path constraint would prune

a larger search space than pruning the union of these paths $E_p = \{a \rightarrow c, b \rightarrow c, c \rightarrow d\}$. It is therefore necessary to develop an algorithm that can decide the minimum set of paths to add to the path constraint.

Algorithm 6.1 STATPATHCONSTRAINTS($G', w, c, m_{best}, \eta, x_{SAT}, \phi$)

```
    // arrival time for nodes and edges are vectors with Monte Carlo values
1   arr_time(v) = ε, arr_time_edge(e) = ε, ∀v ∈ V, ∀e ∈ E'.
    // perform statistical analysis on G'
2   STATANALYSIS(G', w, c, P, C, x_SAT)
    // initialize bit-mask for Monte Carlo iterations that are to be considered
3   B(i) = 1  ∀i ∈ {1, 2, ... # MC iterations}
4   while true
5       p =FINDMAXPATH(G', w, c, m_best, η, x_SAT, φ, B)
6       φ = φ∪ edges(p)
7       l =LENGTH(p)
8       B(i) = B(i) ∧ (l(i) < m_best), ∀i ∈ {1, 2, ... # MC iterations}
        // check if more than η% of the iterations have been covered
9       if COUNT_ONES(B) < η% of # MC iterations
10          break
11  return
```

Algorithm 6.1 embodies a heuristic approach to identify a small set of paths to add to the path constraint. The inputs to the algorithm are the task graph G' to which ordering edges have been added, the performance model w and c, the best makespan found so far m_{best}, the makespan percentile η, the result of the master problem x_{SAT} that encodes the valid schedule found and the (initially empty) list of edges ϕ. The output of the algorithm is the updated list of edges ϕ that stores the paths that together determine the makespan. The first step in the algorithm is to perform a statistical analysis on graph G' using Monte Carlo simulation. This gives us the arrival time for nodes and edges in the graph (line 2). The algorithm then proceeds in a number of iterations(line 4). Each iteration adds a new path to the list of paths that determine the makespan until no more paths are required. The basic idea in the algorithm is to first pick the path that is greater than the best makespan in the maximum percentage of Monte Carlo iterations (line 5). This can be achieved with a traversal of graph G' in reverse topological order using the arrival times computed earlier. The list of edges in the path chosen is then added to ϕ (line 6). the The algorithm then updates a bit-mask that stores the iterations in which this path is smaller than m_{best} (line 8). In case this path is smaller than the best makespan in less than $\eta\%$ of all Monte Carlo iterations (computed through a count

of the bits that are one within the bit-mask), we are done (line 10). If not, we eliminate all Monte Carlo iterations in which this path is greater than the best makespan. Among all other iterations, we again pick the path with the highest fraction of iterations greater than the best makespan, and check if the two paths together now cover $\eta\%$ of all Monte Carlo iterations. If not, we repeat this procedure of eliminating iterations and picking new paths until we do find a set of paths that cover $\eta\%$ of all iterations, where we stop.

Using the above algorithm, we propose the following decomposition-based constraint optimization technique for statistical scheduling. Algorithm 6.2 takes in the task graph G, the performance model w and c, the architecture model (P, C) and the makespan percentile η. The output of the algorithm is the optimized schedule for the task graph onto the architecture. The algorithm follows similar steps as Algorithm 2.4 that optimizes schedules for static models. The parts of the algorithm that deal with the setup and solution of the master problem (lines 1-6), as well as checking the validity of the master problem solution (lines 7-10) are unchanged from Algorithm 2.4. Algorithm 6.2 differs from Algorithm 2.4 in lines 11 and 12, where we compute the makespan of the valid allocation and schedule using a statistical analysis routine (line 11) and use Algorithm 6.1 to add path constraints that eliminate a part of the solution space from consideration in future master problem iterations (line 12).

Algorithm 6.2 STATDA$(G, w, c, P, C, \eta) \rightarrow$ *makespan*

```
1   makespan = ∞
2   φ = empty CNF formula
3   BASECONSTRAINTS(G, w, c, P, C, φ)
4   while (true)
      // master problem
5     x_SAT = SATSOLVE(φ)
6     if (x_SAT = UNSAT)    return makespan
      // sub-problem
7     G' = UPDATEGRAPH(G, x_SAT)
8     if (G' contains a cycle)
9       CYCLECONSTRAINTS(G', x_SAT, φ)
10    else
11      makespan = min{makespan, STATANALYSIS(G') }
12      STATPATHCONSTRAINTS(G', w, c, makespan, η, x_SAT, φ)
```

6.2 Algorithmic extensions

Algorithm 6.2 uses a Boolean Satisfiability based constraint solver to solve the master problem. There is significant opportunity to constrain and guide the solution returned by the SAT solver to improve the efficiency of the decomposition technique. We first present the existing search procedure of the SAT solver, and then show how we can modify this procedure to improve solver performance.

6.2.1 Boolean Satisfiability procedure to solve the master problem

A Boolean Satisfiability based solver uses a branch-and-bound procedure that explores the space of solutions using a series of local scheduling decisions. Each decision involves setting one or more of the Boolean decision variables of the problem to 1 or 0. The branch-and-bound procedure also allows a backtrack from the decisions already made to explore other solutions.

Algorithm 6.3 presents the outline of a general SAT solver presented in [Een and Sörensson, 2003]. The algorithm starts by selecting an unassigned decision variable to a particular value (line 8). All occurrences of the decision variable in the constraints are then replaced by the value chosen. This is called the propagation step, and this may possibly result in more variables being assigned (line 3). The decision phase continues until either all variables are assigned, in which case the problem is satisfiable and the satisfying solution is returned (line 5), or a conflict occurs (line 9). In case there is a conflict, the conflict needs to be analyzed (line 10) and a backtrack is done to the last non-conflicting state (line 14).

The two important steps that can be modified in this procedure are the ANALYZE and DECIDE steps. The ANALYZE step is responsible for identifying whether the set of decisions made so far is incompatible, i.e whether the decisions lead to one or more constraints that are not satisfied. We modify this step by adding an additional pruning step that attempts to identify whether a particular set of decisions can lead to an optimal solution. We describe this procedure in Section 6.2.2. The DECIDE step chooses the next variable to allocate. A SAT solver has an inbuilt mechanism to order variables in a way that gives priority to more active variables. However, as we shall see in Section 6.2.3, it is possible to choose a better ordering to guide the search with the knowledge of the nature of the optimization problem. We do this by using the heuristic list-scheduling decision procedure (Algorithm 5.2) to help guide the next scheduling decision.

Algorithm 6.3 SATSOLVE(ϕ) $\rightarrow$ ({ SAT, UNSAT }, x_{SAT})

```
1   x = list of assignments to variables in φ (initially empty)
2   while ( 1 )
      // propagate variable implications based on current assignments
3     PROPAGATE(φ, x)
4     if (NOCONFLICT(φ, x))
5       if (all variables in x assigned)
6         return (SAT, x)
7       else
          // pick a new decision variable and assign it a value
8         DECIDE(φ, x)
9     else
        // analyze conflict and add a conflict clause
10      ANALYZE(φ, x)
11      if (top level conflict found)
12        return (UNSAT, x)
13      else
          // undo assignments until conflict disappears
14        BACKTRACK(φ, x)
```

6.2.2 Pruning intermediate nodes in the search

The ANALYZE step in Algorithm 6.3 can be modified to invoke the sub-problem at intermediate points in the search. If a partial solution contains a cycle, then any set of future decisions to other variables cannot eliminate the cycle. Therefore, we can prune the search whenever we detect a cycle, and add the corresponding cycle constraint to the problem. We then restart the search and a new iteration of the master problem can commence. If there is no cycle in the partial solution, then we can add ordering edges that captures the partial solution to the task graph. The delay of any path in the graph with ordering edges is a lower bound to the makespan that can be achieved with the partial solution. This can be computed using a statistical analysis based technique. We then check if this lower bound is larger than the best known makespan. If it is indeed larger, then Algorithm 6.1 can be invoked to add the corresponding path constraint. The search is again restarted after the new constraint has been added.

In general, any lower bound to the makespan that is achievable from a partial schedule can be used to facilitate pruning at intermediate nodes of the SAT search. A tight lower bound will identify sub-optimal portions of the search tree earlier in the search process and will hence lead to a faster search. Lower bounds are typically computed on structural properties of the task graphs. The

longest path of the task graph with ordering edges capturing the partial schedule is one example of such a lower bound. Another obvious lower bound is the ratio of the statistical sum of the execution time of all unassigned tasks to the number of processors. More complex lower bounds that attempt to compute the minimum possible communication time between tasks have also been developed for static models [Gerasoulis and Yang, 1992]. These can also be adapted for statistical models by using statistical Monte Carlo analysis.

6.2.3 Guiding master problem search using heuristics

A key part of Algorithm 6.3 is the DECIDE step that picks a new variable and sets it to a value. Although the SAT solver incorporates a generic technique that works well in many cases [Een and Sörensson, 2003], we can tune the policy to target scheduling problems. In a scheduling context, the Dynamic List Scheduling (DLS) heuristic is a good choice for making the next decision. The STATDLSDECIDE procedure used to make a scheduling decision given a partial schedule was described in Algorithm 5.2. Given a partial schedule, the procedure returns the next (task, processor) pair to be scheduled. Such a procedure can directly be invoked on the partial schedules encoded by the intermediate nodes of the SAT search tree, and will return a (task, processor) pair to schedule. The variables corresponding to the allocation of the task to the processor is fixed and the SAT search procedure then resumes. Since DLS has been shown to be a heuristic that produces high-quality solutions [Kwok and Ahmad, 1999a][Davidović and Crainic, 2006], we can expect the search to be directed towards better solutions.

6.3 Iterative decomposition-based algorithm

The algorithm presented in Section 6.1 even with the algorithmic extensions proposed above cannot solve large scale statistical scheduling problems under reasonable time limits. The main problem with the approach is that the lower bound computation at intermediate nodes of the search tree involves a Monte Carlo simulation that is very compute intensive. Since the master problem must use the Monte Carlo analysis at each intermediate node that it explores, it spends a long time in just getting to a leaf node and returning control to the sub problem. We would ideally like single master and sub-problem invocations to complete quickly to allow for a large number of iterations to be executed in a given amount of time.

For task graphs with statistical distributions that are either independent or positively correlated,

and with a required makespan percentile $\eta > 50\%$, then the result of an average case analysis of the task graph can be used as a lower bound. This analysis is a static longest path analysis and does not require any Monte Carlo simulations. In order to perform such an analysis, the task graph corresponding to the partial schedule is modified by replacing all task execution times and communication times with the average value of the respective distributions, and eliminating all edges with a probability of existence less than 50%. For most common distributions, we can also use the median, or the 50^{th} percentile of the execution and communication times rather than the average case values. A linear-time static longest path analysis of this new task graph will then yield a lower bound to the best makespan that can be achieved from the partial schedule. Such a lower bound has previously been used in [Beck and Wilson, 2007] for job-shop scheduling problems. The main problem with such a lower bound is that it is not usually a tight bound, especially at typically used makespan percentiles above 90%. Using a lower bound that is not tight will lead to ineffective pruning of the search space, and will lead to the exploration of sub-optimal parts of the search tree. This can make the algorithm computationally infeasible.

We can modify the above technique in order to tighten the bound obtained while still maintaining the efficiency of computation of the bound. We do this by replacing each statistical variable (task execution times, communication times and dependencies) with samples that are greater than their average case values. In particular, we can opt to sample each distribution at a percentile $\mu > 50\%$. We can then perform a static longest path analysis with these samples to obtain a lower bound estimate. The result of this computation is an increasing function of μ. As we increase μ from 50%, we obtain tighter estimates of the lower bound. However, we cannot increase μ arbitrarily: if we increase it beyond some point, the estimates cease to be lower bounds to the makespan. If we do not use lower bounds, the branch and bound algorithm may end up pruning the portion of the search tree with the optimal solution, and hence lose the guarantee of optimality of the solution. The challenge then is to find the right balance of the sampling percentile μ such that the result of the longest path estimate remains a lower bound and is as tight as possible. Unfortunately, given arbitrary distributions of execution and communication times, it is not possible to analytically obtain the right value for μ.

It is, however, possible to design an algorithm that iteratively explores the space of μ values and guarantees both solution optimality and efficient pruning of sub-optimal solutions. We now describe such an approach. We run the decomposition-based method of Algorithm 6.2 multiple times with decreasing values of μ, starting with μ close to 100% and reducing it to 50%. Each iteration of the algorithm will use the respective μ value to sample the statistical distributions in the task graphs at

intermediate nodes of the SAT search, and use the result of a static analysis on the resulting graph to bound the search in the master problem. In the very first iteration with high μ values, the algorithm will prune most of the solution space (potentially including the the optimal solution) and should return very quickly with a highly sub-optimal solution. Each subsequent iteration will explore a little more of the search space and returns increasing improved solutions. We stop when we reach $\mu = 50\%$. At $\mu = 50\%$, the algorithm is guaranteed to terminate with the optimal solution.

It is important to note that we do not change the nature of the statistical analysis performed at the leaf nodes of the search tree. In particular, the modifications proposed here are only to the lower bound computations in step 10 of the master problem described in Algorithm 6.3. The nature of the sub-problem analysis and the constraints that are added to the master problem remains unchanged from Algorithm 6.2. Therefore all solutions returned by different iterations of the optimization problem are still valid irrespective of the value of μ. It is just the optimality of the solution returned that changes in different iterations. It is possible that the first few iterations of the optimization problem will not yield any solution if we use very high values for μ (close to 100%). In such cases, we can ignore such iterations and continue by lowering the value of the μ parameter.

The iterative algorithm has an important practical advantage over a branch and bound algorithm using a single value of $\mu = 50\%$. When time limits are imposed on the scheduling approach, we find that a branch and bound algorithm with $\mu = 50\%$ fails to make significant progress in the search. In our experiments, we found that almost no improvement was made over a heuristic solution under a time limit of 15 minutes. However, with an iterative approach, the early iterations of the algorithm with high values of μ tend to terminate quickly. Although the algorithm does not guarantee optimal solutions in these early iterations, it does explore many leaf nodes over the course of the search. In practice, we found that it was often the case that the optimal solution was obtained at early iterations with μ values around 80-90%. Under such circumstances, the iterative approach finds the optimal solution fairly early in the search and spends the rest of the time proving the optimality of the solution. It can therefore return good solutions even under strict time limits.

The iterative scheme is also superior to the algorithm described in Section 6.1. The algorithm in Section 6.1 spends most of its time computing lower bounds at intermediate search nodes. The iterative algorithm, on the other hand, explores many leaf nodes and therefore identifies better solutions to the optimization problem. We will quantify the difference in the quality of results obtained using these two algorithms in Section 6.4.1.

6.4 Results

In this section, we compare the results of the statistical decomposition-based algorithm (DA) described in Section 6.1 with the iterative algorithm (IT-DA) that uses static analysis for pruning as described in Section 6.3. We show that the iterative IT-DA algorithm performs better than the DA algorithm in terms of the optimality of the solution achieved in a constant runtime. We then compare the iterative IT-DA algorithm with the statistical Dynamic List Scheduling (DLS) technique and statistical Simulated Annealing (SA) techniques described in Chapter 5.

6.4.1 Comparison of different decomposition-based scheduling approaches

We compare the DA algorithm with the iterative IT-DA algorithm using the random task graph instances of Table 5.4. Figure 6.2 shows the average percentage improvement of the makespan results computed by the DA and IT-DA algorithms over the statistical DLS solution. All algorithms optimize for the makespan percentile $\eta = 95\%$. Each data point is computed by scheduling a task graph with 50 tasks on 4,8 and 16 processors and taking the average percentage difference of the two algorithms from the DLS solution. The IT-DA algorithm is run using sampling percentiles μ of 99%, 95%, 90%, 80% and 50%. Each of these algorithms was allowed to run for a total time of 10 minutes. In the IT-DA algorithm, the iteration corresponding to a lower μ value was started only if the previous iteration with a higher μ completed within the timeout. The graph in Figure 6.2 plots the average percentage differences from the statistical DLS solution versus edge density of the task graph instances. The graph shows that the iterative DA algorithm consistently improves the DLS makespan more than the DA approach within the timeout of 10 minutes. This is because the iterative DA algorithm quickly reaches feasible solutions at high sampling percentiles μ, and slowly improves the result on decreasing μ. On the other hand, the DA technique spends much of its time in computing statistical lower bounds that are used in pruning nodes in the search tree, and hence it explores only a small portion of the search space.

The IT-DA algorithm is superior to the DA algorithm for most scheduling instances. The DA algorithm typically fails to make any improvement to the statistical DLS algorithm, and is therefore of limited use in a toolbox of statistical scheduling approaches. We only consider the IT-DA algorithm in the rest of our comparisons.

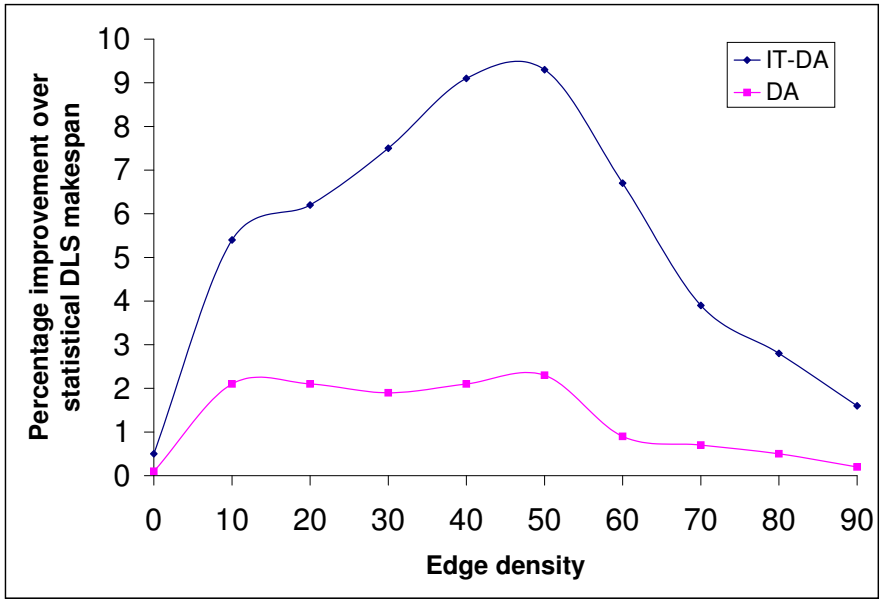


Figure 6.2: Average percentage improvement of the makespans obtained by the DA and IT-DA statistical decomposition approaches over the statistical DLS makespan for random task graphs containing 50 tasks scheduled on 4. 8 and 16 fully-connected processors as a function of edge density.

6.4.2 Comparison of decomposition-based scheduling to other approaches

We now compare the scheduling results of the iterative DA approach with the statistical DLS and SA approaches described in Chapter 5. The key criteria for comparison are the quality of the solution obtained, the runtime of the algorithm and the extensibility of the algorithm. We evaluate the extensibility of the scheduling approach by the ease with which additional constraints can be added to the problem and the quality of solution of the resulting algorithm. In particular, we try our scheduling methods on different architectural topologies and study the quality of the solutions obtained on these varied topologies.

Table 6.1 shows the makespan obtained by scheduling task graphs corresponding to the IPv4 packet forwarding application on 8 processors that are completely connected (Full) and connected using a ring topology (Ring). Recall that in Table 5.2 of Chapter 5, we replicated the task graph for IPv4 forwarding from 1 to 10 times to correspond to forwarders with different numbers of channels. We perform the same replication in Table 6.1 as well. Column 1 in Table 6.1 shows the number of tasks in each of these instances. The other columns show the makespan obtained using the three algorithms at two different makespan percentiles of 99% and 95%. The IT-DA algorithm was run

using static percentiles of 99%, 95%, 90%, 80% and 50%. The total time allowed for a single IT-DA run was 15 minutes. All IT-DA runs that complete within the time limit with the optimal solution are highlighted in bold.

The results of Table 6.1 show that the IT-DA results are within 4% of both the SA and DLS solutions for the 99^{th} and 95^{th} percentiles on the fully connected topology. This is true even when the IT-DA algorithm terminates with the optimal solution (this happens when there are fewer than 60 tasks). This means that the DLS and SA solutions yields near-optimal solutions for fully connected topologies. A similar trend for the makespans achieved through various algorithms has already been noted in the context of static models (Chapter 2). The statistical DLS and SA algorithms are essentially derived from the algorithms for static models, and it is therefore not surprising to obtain a similar quality of results in the statistical versions as the static versions.

	Full						Ring					
	η = 99%			η = 95%			η = 99%			η = 95%		
# Tasks	DLS	SA	IT-DA	DLS	SA	IT-DA	DLS	SA	IT-DA	DLS	SA	IT-DA
15	135	135	**135**	105	105	**105**	145	135	**135**	105	105	**105**
28	135	135	**135**	125	125	**120**	145	145	**135**	130	125	**120**
41	135	135	**135**	135	135	**135**	175	150	**145**	155	155	**140**
54	160	160	**155**	145	135	**135**	240	185	175	230	210	195
67	185	170	175	175	160	175	290	230	230	250	225	215
80	195	185	195	190	185	190	365	285	295	285	260	265
93	210	220	210	225	215	215	360	280	295	315	275	275
106	250	240	225	230	200	205	365	340	325	400	360	370
119	255	270	255	255	260	255	395	395	370	395	370	370
132	290	270	280	285	270	270	410	375	390	440	360	350
Avg. % diff. from DLS	-	1.3	1.6	-	3.9	3.0	-	12.5	13.9	-	7.9	10.1

Table 6.1: Makespan results for the statistical DLS, SA and iterative DA methods on task graphs derived from IPv4 packet forwarding scheduled on the 8 processors connected in full and ring topologies.

The statistical DLS algorithm is not as optimal when the architectural topology is a ring of processors. From Table 6.1, we find that the DLS makespan is on average more than 10% off of the decomposition approach. This helps demonstrate one of the key issues with heuristics - they tend to be difficult to extend to accommodate additional constraints, and cannot be expected to produce results of high quality once such extensions have been made. In contrast, we note that the statistical SA algorithm seems to extend well when we modify the topology - the difference between the SA and IT-DA approaches remains small on both topologies.

The trends are similar for other architectural topologies. Figures 6.3 and 6.4 show the percentage improvement of the statistical SA and IT-DA makespans over the DLS makespan for the IPv4 application scheduled on the architectures of Figure 2.10(b) and (c) respectively. The differences are shown for different IPv4 instances with the task graphs unrolled from 1 to 10 times. From the two figures, we can see that the statistical DLS approach can yield makespans that are up to 35% worse than the SA or iterative DA approaches. We can also see that the SA and iterative DA approaches yield similar results. While the iterative DA approach often gives better results for small task graphs, the SA approach becomes better at large task graphs. This trend was also seen in Chapter 2 for static scheduling. This can be attributed to the "knee" effect shown by constraint optimization methods, wherein problems up to a certain size are solved quickly, and problems beyond that size take a very long time to solve. The scheduling problem is known to be hard to solve optimally, and hence this is an expected trend.

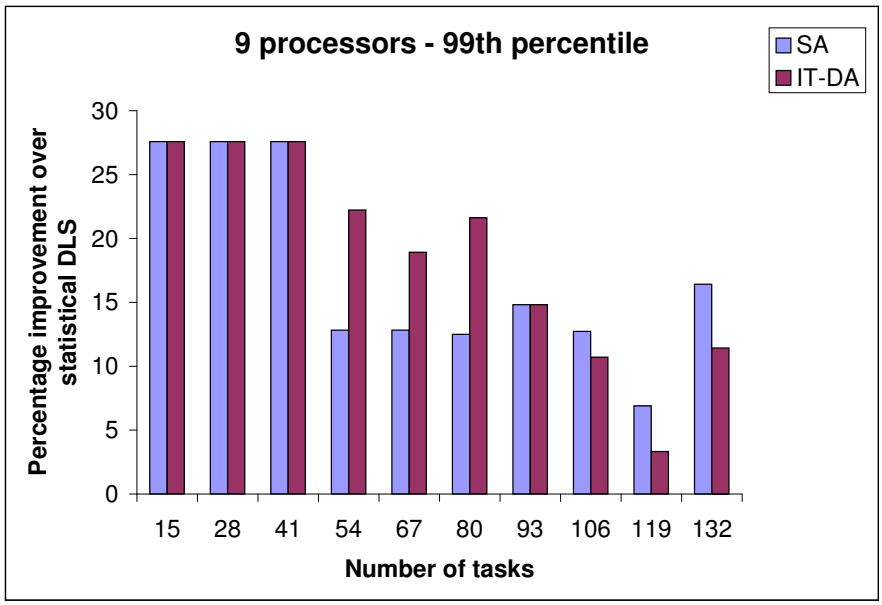


Figure 6.3: Percentage improvement of the statistical SA and IT-DA makespans over the statistical DLS makespan for task graphs derived from the IPv4 packet forwarding application scheduled on the architecture of Figure 2.10(b).

The runtime of the three approaches is also a useful comparison metric. Table 6.2 shows the runtime of the three algorithms for the corresponding entries of Table 6.1. The table shows that the statistical DLS heuristic completes within 2 minutes for all scheduling instances. The SA algorithm finishes within 10 minutes even at the largest instances. The iterative DA algorithm, on the other

# Tasks	Full						Ring					
	$\eta = 99\%$			$\eta = 95\%$			$\eta = 99\%$			$\eta = 95\%$		
	DLS	SA	IT-DA	DLS	SA	IT-DA	DLS	SA	IT-DA	DLS	SA	IT-DA
15	0	0	0	0	0	0	0	0	0	0	0	0
28	0	0	1	0	0	1	0	0	0	0	0	1
41	0	1	2	0	1	2	0	1	2	0	1	2
54	0	2	3	0	2	4	0	1	15	0	2	15
67	0	2	15	0	2	15	0	2	15	0	2	15
80	0	4	15	0	3	15	0	2	15	0	3	15
93	0	4	15	0	4	15	1	4	15	0	4	15
106	0	5	15	0	5	15	1	6	15	0	5	15
119	1	7	15	1	6	15	1	7	15	1	6	15
132	1	8	15	1	6	15	1	7	15	1	8	15

Table 6.2: Runtime (in minutes) for the statistical DLS, SA and iterative DA methods on the mapping instances of Table 6.1.

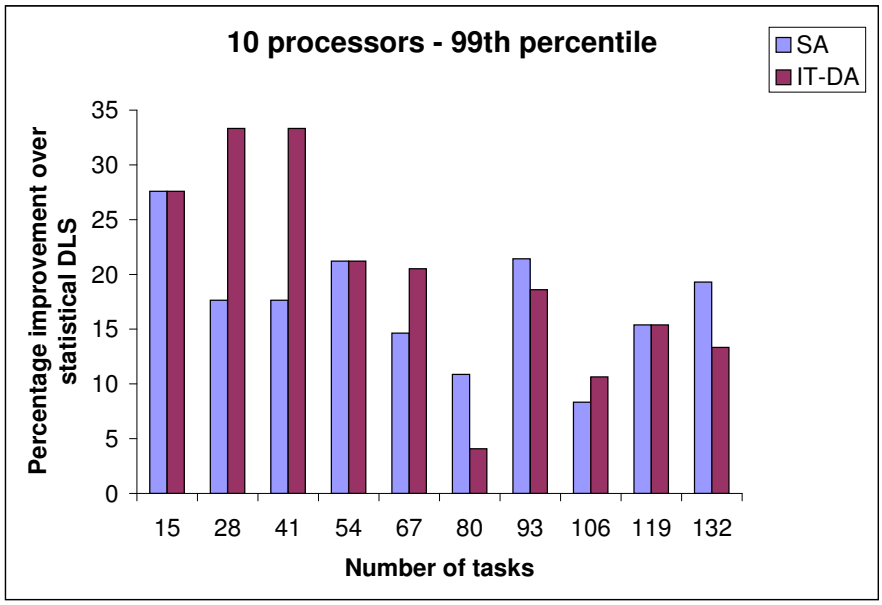


Figure 6.4: Percentage improvement of the statistical SA and IT-DA makespans over the statistical DLS makespan for task graphs derived from the IPv4 packet forwarding application scheduled on the architecture of Figure 2.10(c).

hand, does not finish execution beyond 60 tasks, and times out at 15 minutes. For instances below 60 tasks, the iterative DA algorithm terminates with the optimal solution.

The choice of scheduling methods depends on the desired trade-off between the quality of solution and runtime. We find that the statistical DLS algorithm performs very well on certain types of scheduling problems. These include problems where there are no additional constraints such as

architecture topologies added to the scheduling problem. Another set of scheduling instances where a DLS based algorithm does well is when the number of edges in the task graph is either close to zero or close to the maximum possible number of edges. Table 6.2 shows that the exact IT-DA algorithm does not improve the DLS solution at zero and high edge densities. This can also be seen in decoding H.264 sequences that consist mainly of P-macroblocks. We showed in Chapter 5 that the statistical DLS procedure shows less than 8% performance degradation compared to statistical SA for such sequences. We find that the IT-DA algorithm does not improve the DLS result for this application. For these examples where the DLS solution is near-optimal, it is clearly the solution of choice, as it take less than 2-3 minutes to execute even for large task graphs as compared to 20 minutes or more for the other methods. For other problems that have additional constraints on mapping and have an intermediate number of edges in the task graph corresponding to the application (the IPv4 application is a good example), the DLS makespan can be significantly suboptimal. For such scheduling instances, either the SA or IT-DA algorithms can be used. For statistical scheduling, we generally find the SA and IT-DA algorithms to yield similar makespans. However, the statistical SA scheme usually scales better to larger task graphs. In our experiments, we found the SA algorithms to produce better schedules than the IT-DA algorithm for task graphs of size greater than 100 tasks. Thus for large task graphs, the statistical SA algorithm offers the best combination of quality of results and runtime.

Chapter 7

Conclusions

Single-chip multiprocessors have now become prevalent in both the embedded and general-purpose markets. The challenge before the programmer is to productively program these multiprocessors so as to use the parallelism available in the devices. This motivates the development of automated tools for parallel application deployment and design space exploration. A key step in such an automated flow is mapping task-level concurrency present in the application to the parallel hardware resources in the multiprocessor platform.

In this dissertation, we concentrated on investigating and evaluating approaches to compile-time scheduling of applications with task-level concurrency onto multiprocessors. Such techniques rely on the availability of realistic models of the application and architecture at compile time. Compile-time scheduling techniques incur minimal overhead while running the application. They are also useful in rapid design space exploration of micro-architectures and systems. Compile-time methods are not applicable if there is no knowledge of the parallel tasks and application workload at compile time. In that case, run-time or dynamic scheduling techniques must instead be used.

Scheduling problems that arise in realistic application deployment and design space exploration frameworks can encompass a variety of objectives and constraints and require different features to be exposed in the application and architecture models. In order for scheduling techniques to be useful for realistic exploration frameworks, they must therefore be sufficiently flexible to be applied to a range of problems. They must also produce high quality solutions and must be computationally efficient. In this dissertation, we compared the use of heuristics, simulated annealing and exact optimization methods for solving three different scheduling problems in view of the above metrics. All the problems that we considered involved scheduling tasks onto parallel architectures, but differed in terms of the constraints and optimization objective of the scheduling problem. The

diversity of problems studied gave us a base for studying the extensibility of scheduling methods. It also helped provide a more holistic view of the merits of different scheduling approaches in terms of their efficiency and quality of solutions produced on scheduling problems in general.

We now briefly summarize the results of this dissertation and comment on directions for future work.

7.1 Comparison of scheduling approaches

In this work, we investigated and evaluated different techniques for scheduling task-level concurrency in applications onto multiprocessors. Scheduling techniques broadly fall under one of three categories: heuristics, randomized algorithms and exact approaches. In this work, we picked one interesting candidate from each category for each scheduling problem we considered. These approaches offer different trade-offs in terms of solution quality, optimality of solution and flexibility. As a general strategy, heuristics are most useful when they have been tuned to a particular optimization problem. A programmer may find that the optimization problem under consideration fits exactly into a known heuristic solution. In such a case, heuristics are often a good choice. However, for many practical problems, this may not be the case. A more general technique for solving optimization problems is then required. Randomized methods and exact approaches both offer flexibility in terms of the range of problems they can accommodate. Exact approaches additionally offer the promise of obtaining optimal solutions to the scheduling problem, but usually succeed only on problems of small to moderate size in terms of the number of variables and constraints in the problem. For large and complex optimization problems, randomized techniques are the method of choice.

We will now discuss the use of heuristics, a simulated-annealing based randomized algorithm and a constraint-optimization based exact approach to solve different scheduling problems. We will highlight the advantages and disadvantages of using each approach.

7.1.1 Heuristic techniques

Heuristic methods are computationally efficient and can handle large scheduling problems with thousands of tasks. Consequently, they are a useful option to have in a toolbox of scheduling methods to be used when other techniques fail to make progress. However, heuristics do not offer the promise of obtaining optimal solutions to the scheduling problem or even bounds on the optimality

of the solution achieved. In this work, we found that the quality of solutions produced by heuristics can differ depending on the exact scheduling problem. In Chapter 2, we found that a dynamic list scheduling heuristic can yield solutions within 5-10% of the optimal when scheduling concurrent tasks onto regular homogeneous processors. On the other hand, heuristics for other scheduling problems can be as much as 40-50% off from the optimal solution. One example is the data transfer minimization problem of Chapter 3. The unpredictability of heuristics is a big disadvantage to their use in practical scheduling frameworks.

Another key drawback of heuristic solutions is the lack of flexibility to accommodate different sets of models and objectives. As an example, the problems of both Chapters 2 and 3 involve finding the optimal ordering of tasks within a processor. However, since the models and objectives for the two problems are different, they use very different heuristics. The list scheduling heuristics for the task allocation and scheduling problem of Chapter 2 are primarily focused on prioritizing the execution of tasks on the "critical path" of the application based on task execution times. This is quite different from the heuristics used for the task and data transfer scheduling problem of Chapter 3, which does not model task execution times. A recently proposed algorithm for this problem instead relies on a depth-first order to minimize the amount of data that is live at any point in the schedule.

Even when we consider a single scheduling problem, it is often hard for heuristics to handle the presence of additional constraints in a scheduling problem without losing their efficiency. As an example, the dynamic list scheduling (DLS) heuristic, which is a popular heuristic for allocating and scheduling tasks to multiprocessors, was primarily developed for mapping task graphs onto fully connected homogeneous architectures. In Chapter 2, we showed that it produces very high quality solutions that are within 5-10% of the optimal solution on such architectures. However, when mapping tasks onto more irregular and heterogeneous architectures, the heuristic must be extended. The extended heuristic produces less optimal results; it can be up to 25% worse than the optimal solution. Moreover, the very fact that the heuristic must be manually modified for each possible constraint detracts from its value for general scheduling problems. Therefore we need a method that can more flexibly handle constraints such as heterogeneity, memory size limits, preferred task allocations or task clustering.

7.1.2 Constraint Optimization methods

Constraint optimization techniques typically use an underlying solver technology such as Mixed Integer Linear Programming (MILP) or Boolean Satisfiability (SAT) solvers to specify the constraints and optimization objective of the optimization problem. The range of optimization objectives and side constraints that can be added to a scheduling problem is only limited by the structure of the constraints that the solver technology can handle. In addition to Boolean clauses in SAT and linear inequalities in an MILP solver, solvers that allow pseudo-boolean constraints, sat-modulo constraints and constraints in many other specialized "theories" have become popular in recent years. We were able to use constraint optimization based techniques to solve the three different scheduling problems we considered.

Of the three classes of scheduling algorithms, constraint optimization techniques are the only ones that offer the promise of obtaining optimal solutions to the scheduling problem when they are given enough time to execute. It is often the case that optimal solutions cannot be guaranteed when constraint optimization methods are limited to a runtime of up to 30 minutes. In our experience, even when solvers are allowed to run for a few hours, there is usually only slight improvement in the quality of the solutions returned. However, even under such conditions, constraint optimization techniques can provide bounds as to how much the result is inferior to the optimal solution. Constraint optimization techniques also have the useful property that the solutions returned by these techniques are guaranteed to improve with runtime; hence they allow for tradeoffs between runtime and quality of the solution.

However, constraint solvers suffer from the point of view of the efficiency metric: they cannot usually handle large scheduling instances. The solution space of scheduling problems is not smooth and is difficult to explore completely. Such solvers usually have a "knee" effect: they handle all problems up to a certain size well, but suddenly fail at a certain problem size. For the task allocation and scheduling problem of Chapter 2, single pass MILP approaches can only schedule task graphs of up to 30-50 tasks, which corresponds to about a thousand variables and constraints. For the scheduling problem in Chapter 3, an MILP based-approach could only schedule task graphs with up to 30 tasks and data items, which again involves a few thousand variables and constraints. This can be improved by the use of improved problem formulations or the use of more powerful solvers. However, such improvements can only push the "knee" to larger problem sizes; the ultimate exponential nature of the runtime of such solvers will persist. For the task allocation and scheduling problem of Chapter 2, we were able to decompose the scheduling problem in order to accelerate

the solver search mechanism. By using such a decomposition technique, we were able to push the frontier of the number of tasks that we could schedule from less than 50 to over 150 tasks, which corresponds to a few tens of thousands of variables and constraints. However, constraint-based techniques are still not effective beyond that point.

7.1.3 Simulated Annealing

Simulated Annealing is a randomized algorithm that is effective on large-scale scheduling problems. It offers a middle ground between heuristics and exact techniques. The algorithm does not offer the promise of obtaining exact solutions, but does allow for some tradeoff between computation time and the quality of solutions they produce. One of the big drawbacks of simulated annealing algorithms has been the perceived randomness and potential sub-optimality of the solution they end up with. While it is true that the result of simulated annealing does vary somewhat across different runs, we found that such differences were usually minor and did not affect the quality of the solution. Further, we found that it was possible to tune the parameters to the simulated annealing technique in order to produce high quality solutions. From our experiments on all three scheduling problems, we found that simulated annealing produced solutions that were within 20-25% of the solutions returned by exact methods for small problems. For larger problems, where constraint optimization approaches did not make much progress towards solving the problem, simulated annealing techniques yielded better solutions than constraint programming methods. Simulated Annealing also gave better results than heuristics in almost all our problems. The difference varied from a low of 5% to a high of 50% depending on the exact scheduling problem considered.

Simulated Annealing methods also scale well to large scheduling problems. In Chapter 3, we were able to solve large scheduling instances of up to 1600 tasks using simulated annealing methods. For the other scheduling problems, we were able to handle instances with a few hundred (200-500) tasks, which would correspond to about a hundred thousand variables and constraints. Simulated annealing can solve large enough problems to be of use in solving real-world scheduling problems.

An additional advantage of simulated annealing is that it is very extensible. The algorithm itself is a meta-optimization procedure that can be tuned to different optimization problems with a variety of parameters. In particular, the COST function that evaluates the optimization function corresponding to a "state" (corresponding to a particular schedule) is a callback function that can easily be tuned to different problems. Different constraints to the scheduling problem can also be easily incorporated as a part of deciding whether a state represents a valid or invalid schedule. We

were able to use simulated annealing based techniques for all three of our scheduling problems by changing the parameters of the problem.

7.2 Incorporating Variability into Compile-time scheduling models and methods

One of the key criticisms of compile-time scheduling methods is that the application workload or execution characteristics of applications may not be fully known at compile-time, due to data-dependent executions or jitter in execution times. Consequently, such methods usually optimize for worst-case or common-case application behavior depending on whether the application has real-time constraints or not. For many applications that have soft real-time constraints, neither of these optimize for the right metric of application performance. The performance metric for soft real-time applications is the time at which a given percentile of all inputs complete. For instance, a packet forwarding application may require a service guarantee that 95% of all input packets complete in a given time frame. In such a case, the 95^{th} percentile of the application execution time is the correct metric of system performance. In general, the value of the percentile is a user-provided parameter that must be taken into account during the optimization.

In this work, we extend compile-time scheduling models and methods to optimize soft real-time applications. We adopt a profiling based approach to capture the variability in application execution time. We use statistical variables to represent task dependencies and performance characteristics. We then extend our scheduling techniques to handle such variations. Our scheduling techniques take into account the percentile of the makespan to optimize for.

We believe that statistical scheduling techniques will be of high relevance for future applications and architectures. The complexity of applications in terms of the amount of control flow has in general been increasing in many fields. A typical example has been in video codecs where the trend has been towards more efficient but logically complex coding schemes such as H.264 [Davare, 2007]. The performance characteristics of such applications tend to be highly dependent on input data characteristics. For the sake of the accuracy of performance modeling, it is necessary for the variability in such applications to be captured. This work has helped illustrate some of the basic models and methods that are used for scheduling such applications.

7.3 Future Work

The current work can be extended in many different directions. The first direction is in trying our scheduling techniques on a broader set of applications. The second direction is in exploring other compile-time scheduling approaches from the broad category of evolutionary algorithms. The third direction is in exploiting the symmetry present in common scheduling problems in order to improve the computational efficiency of our solution techniques. The fourth direction is in efficiently implementing our scheduling methods on current day parallel platforms. Finally, for a complete solution to the mapping problem, we must also include dynamic scheduling approaches which are applicable when we do not have knowledge about the application workload at compile-time.

7.3.1 Broader range of applications

In this work, we limited our experiments to moderate-sized applications from the fields of networking, video processing and machine learning. However, we believe that the main impact of this work will be felt in more complex applications, where a manual mapping effort will likely prove very difficult. Therefore, more complex applications, especially from emerging workload classes such as the recognition, mining and synthesis benchmark [Chen *et al.*, 2008] should be tried.

7.3.2 Other evolutionary algorithms

The area of evolutionary algorithms comprises a fairly broad set of algorithms including simulated annealing, genetic algorithms, genetic programming, ant-colony optimization and others. We picked the simulated annealing algorithm for our study since this is a technique that has been used in a variety of scheduling problems. In view of the successful application of simulated annealing for the problems we studied, it will be of great interest to study the use of other such algorithms. Some algorithms such as ant-colony optimization have been recently successfully applied to one resource-constrained scheduling problem [Wang *et al.*, 2007]. It will be of interest to see how extensible the technique is to other scheduling problems.

7.3.3 Symmetry considerations

Scheduling problems tend to have symmetry either in the task graph or in the architecture model. Our existing solver techniques do not take into account the symmetry to reduce the effective solution space of the problem. They instead explore many symmetric solutions over the course of

the search. One approach to exploit symmetry is to avoid exploring certain schedules if symmetric schedules have already been explored. This requires keeping a history of already explored states. A better alternative is to encode the problem in such a way as to resolve symmetries. Such an approach can significantly help boost solver performance.

7.3.4 Parallel Implementation of Scheduling Methods

The advent of many-core machines into general-purpose computing means that most scheduling algorithms will likely run on a parallel machine. In this context, we can improve solver efficiency by parallelizing the solver to use multiple cores. Since simulated annealing and constraint optimization methods are the most promising methods in terms of solution quality, they are the key targets for parallel execution.

A simulated annealing algorithm works by making a number of transitions at a given temperature and evaluating a cost metric on each of them. Each of these transitions is then accepted or rejected depending on the result of the cost evaluation. After a certain number of such transitions are performed, the temperature is then modified, and new transitions are evaluated at the new temperature. The set of transitions at a given temperature can be performed and evaluated in parallel. However, the temperature update depends on the results of these evaluations, and hence must wait until all transitions at the temperature complete.

In the simulated annealing algorithms in Chapters 2 and 5, there is a parameter L that determines the number of transitions to be made at a single temperature. The value of L in our experiments was 500. Thus there is a 500-fold parallelism to be exploited in simulated annealing. Since each of these transitions are completely independent of each other, it should be relatively easy to exploit the parallelism. Simulated Annealing should then scale in performance with the number of cores on future many-core devices. We believe this is an core additional advantage that will make simulated annealing a very competitive tool as scheduling is relocated to many-core machines.

In contrast, branch-and-bound and related constraint solver methods are not so easily parallelized, and their parallelization is a topic of active research [Chu *et al.*, 2008] [Gil *et al.*, 2008] [Hamadi *et al.*, 2008]. One of the best performing parallel SAT solvers works by running different SAT solvers simultaneously and picking the result of the first solver that completes [Hamadi *et al.*, 2008]. It is unclear if such techniques will scale to dozens or hundreds of cores. Additional research in this field is highly important if constraint solvers are to remain viable on many-core platforms.

7.3.5 Dynamic Scheduling

This dissertation has focused entirely on compile-time methods for scheduling. Compile-time methods are useful when there is knowledge at compile time about the tasks and execution times (either deterministically or statistically). However, in certain applications, the tasks that need to be performed will depend on the inputs. In such cases, dynamic scheduling approaches are the alternative. The criteria for choosing dynamic scheduling algorithms are different from the ones for compile-time scheduling. Here the main aim is to minimize scheduling overhead; hence it is essential to use heuristics. In fact, even approaches like the dynamic list scheduling algorithm discussed in Chapter 2 may be too computationally expensive. A common dynamic scheduling algorithm is based on maintaining a local work-queue of tasks to be executed on each processor [Casavant and Kuhl, 1988]. The principle behind this method is that each processor preferentially executes tasks from its local queue. The execution of a task can lead to more tasks being enqueued; these are appended onto the local task queue. It may so happen that the task queue for some processors becomes empty; in this case some work is "stolen" from other processors based on a greedy heuristic scheme. This is typically called a "work-stealing" approach and is used in many on-line scheduling algorithms [Rudolph *et al.*, 1991] [Blumofe and Leiserson, 1999]. Sgall et al. conduct a more intensive survey of on-line scheduling methods [Sgall, 1997].

7.4 Summary

The goal of this research was to identify and evaluate techniques to solve compile-time scheduling problems that arise in practical parallel application deployment scenarios. Over the course of this dissertation, we identified three broad classes of scheduling approaches: heuristics, randomized/evolutionary algorithms and exact constraint optimization techniques that can be used to solve such scheduling problems. Since exact approaches offer the promise of optimal solutions to scheduling problems, they were of key interest to us. One of our research objectives was in improving exact scheduling approaches to a point where they could solve problems of practical size. Our results here have been mixed. Over the course of this research, we improved the scale of the problems that exact approaches could handle by an order of magnitude (in terms of the number of variables and constraints handled). While we believe that exact approaches are now at a point where they can be used for small problems of practical interest, they still cannot handle large problems. For large and complex design spaces, randomized techniques have proved to be the method of choice.

Bibliography

[Agarwal *et al.*, 1988] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache Performance of Operating Systems and Multiprogramming Workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, Nov 1988.

[Altera Inc., 2003] Altera Inc. *System-on-Programmable-Chip (SOPC) Builder*. Altera Corporation, user guide version 1 edition, June 2003.

[Asanovic *et al.*, 2006] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, University of California at Berkeley, Dec 2006.

[Atamtürk and Savelsbergh, 2005] Alper Atamtürk and Martin W.P. Savelsbergh. Integer Programming Software Systems. *Annals of Operations Research*, 140:67–124, 2005.

[Baker, 1995] F. Baker. *Requirements for IP Version 4 Routers*. Network Working Group, Request for Comments RFC-1812 edition, June 1995.

[Bambha and Bhattacharyya, 2002] Neal K. Bambha and Shuvra S. Bhattacharyya. System Synthesis for Optically Connected Multiprocessors on Chip. In *Proc. of the International Workshop for System on Chip*, 2002.

[Beck and Wilson, 2007] J. Christopher Beck and Nic Wilson. Proactive Algorithms for Job Shop Scheduling with Probabilistic Durations. *Journal of Artificial Intelligence Research*, 28:183–232, 2007.

[Bender, 1996] Armin Bender. MILP Based Task Mapping for Heterogeneous Multiprocessor Systems. In *Proceedings of EDAC*, pages 283–288, 1996.

[Benders, 1962] Jacques F. Benders. Partitioning Procedures for Solving Mixed-Variables Programming Problems. *Numerische Mathematik*, 4(1):238–252, Dec 1962.

[Benini *et al.*, 2005] Luca Benini, Davide Bertozzi, Alberto Guerri, and Michela Milano. Allocation and Scheduling for MPSoCs via Decomposition and No-Good Generation. In *Principles and Practice of Constraint Programming, 11th International Conference*, pages 107–121, 2005.

[Bleiweiss, 2008] Avi Bleiweiss. GPU accelerated pathfinding. In *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 65–74, 2008.

[Blumofe and Leiserson, 1999] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[Boeres and Rebello, 2003] Cristina Boeres and Vinod E. F. Rebello. Towards optimal static task allocation and scheduling for realistic machine models: Theory and practice. *International Journal of High Performance Computing Applications*, 17(2):173–189, 2003.

[Bokhari, 1981] Shahid Hussain Bokhari. On the Mapping Problem. *IEEE Transactions on Computing*, C-30(5):207–214, 1981.

[Borkar, 1999] Shekhar Borkar. Design Challenges of Technology Scaling. *IEEE Micro*, 19(4):23–29, July-August 1999.

[Brucker, 2001] Peter Brucker. *Scheduling Algorithms*. Springer-Verlag New York, Inc., 3rd edition, 2001.

[Canny, 1986] J Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, 1986.

[Casavant and Kuhl, 1988] Thomas L. Casavant and Jon G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, 1988.

[Chen *et al.*, 1994] William Y. Chen, Scott A. Mahlke, Nancy J. Warter, Sadun Anik, and Wen mei W. Hwu. Profile-assisted instruction scheduling. *International Journal on Parallel Programming*, 22(2):151–181, 1994.

[Chen *et al.*, 2008] Yen-Kuang Chen, J. Chhugani, P. Dubey, C.J. Hughes, Daehyun Kim, S. Kumar, V.W. Lee, A.D. Nguyen, and M. Smelyanskiy. Convergence of Recognition, Mining, and Synthesis Workloads and Its Implications. *Proceedings of the IEEE*, 96(5):790–807, May 2008.

[Chong *et al.*, 2007] Jike Chong, Nadathur Rajagopalan Satish, Bryan Catanzaro, Kaushik Ravindran, and Kurt Keutzer. Efficient Parallelization of H.264 Decoding with Macro Block Level Scheduling. In *2007 International Conference on Multimedia and Expo: ICME 2007*, pages 1874–1877, July 2007.

[Chu *et al.*, 2008] Geoffrey Chu, Aaron Harwood, and Peter J. Stuckey. PMiniSat - A parallelization of MiniSat 2.0, Mar 2008. http://www-sr.informatik.uni-tuebingen.de/sat-race-2008/descriptions/solver%5F32.pdf.

[Coffman, 1976] Edward G. Coffman. *Computer and Job-Shop Scheduling Theory*. John Wiley and Sons, Inc., New York, February 1976.

[Cucchiara *et al.*, 1999] Rita Cucchiara, Massimo Piccardi, and Andrea Prati. Exploiting Cache in Multimedia. In *1999 IEEE Conference on Multimedia Systems*, pages 345–350, July 1999.

[Davare *et al.*, 2006] Abhijit Davare, Jike Chong, Qi Zhu, Douglas Michael Densmore, and Alberto Sangiovanni-Vincentelli. Classification, Customization, and Characterization: Using MILP for Task Allocation and Scheduling. Technical Report UCB/EECS-2006-166, EECS Department, University of California, Berkeley, Dec 2006.

[Davare, 2007] Abhijit Davare. *Automated Mapping for Heterogeneous Multiprocessor Embedded Systems*. PhD thesis, University of California at Berkeley, Sep 2007.

[Davidović and Crainic, 2006] Tatjana Davidović and Teodor Gabriel Crainic. Benchmark-Problem Instances for Static Scheduling of Task Graphs with Communication Delays on Homogeneous Multiprocessor Systems. *Computers & OR*, 33:2155–2177, 2006. http://www.mi.sanu.ac.yu/∼tanjad/tanjad_pub.htm.

[Davidović *et al.*, 2004] Tatjana Davidović, Leo Liberti, Nelson Maculan, and Nenad Mladenović. Mathematical Programming-Based Approach to Scheduling of Communicating Tasks. Technical Report G-2004-99, Cahiers du GERAD, December 2004.

[Devadas and Newton, 1989] Srinivas Devadas and Arthur Richard Newton. Algorithms for Hardware Allocation in Data Path Synthesis. *IEEE Transactions on Computer-Aided Design*, 8(7):768–781, July 1989.

[Dick *et al.*, 2003] Robert P. Dick, David L. Rhodes, Keith S. Vallerio, and Wayne Wolf. TGFF: Task Graphs for Free (TGFF v3.0), Aug 2003. http://ziyang.ece.northwestern.edu/tgff/.

[Dutertre and de Moura, 2006] Bruno Dutertre and Leonardo de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Proceedings of the 18th Computer-Aided Verification conference*, volume 4144 of *LNCS*, pages 81–94. Springer-Verlag, 2006.

[Eatherton, 2005] W. Eatherton. The Push of Network Processing to the Top of the Pyramid. keynote address at the Symposium on Architectures for Networking and Communication Systems, October 2005. http://www.cesr.ncsu.edu/ancs/slides/eathertonKeynote.pdf.

[Een and Sörensson, 2003] Niklas Een and Niklas Sörensson. An Extensible SAT-solver [ver 1.2]. In E. Giunchiglia and A. Tacchella, editors, *Lecture Notes in Computer Science*, volume 2919 of *SAT*, pages 502–518. Springer, 2003.

[Ekelin and Jonsson, 2000] Cecilia Ekelin and Jan Jonsson. Solving Embedded System Scheduling Problems using Constraint Programming. In *IEEE Real-Time Systems Symposium*, Orlando, Florida, USA, Nov 2000.

[El-Rewini *et al.*, 1995] Hesham El-Rewini, Hesham H. Ali, and Ted Lewis. Task Scheduling in Multiprocessing Systems. *Computer*, 28(12):27–37, 1995.

[Farach and Liberatore, 1998] Martin Farach and Vincenzo Liberatore. On local register allocation. In *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, pages 564–573, 1998.

[Fernandez and Bussell, 1973] E.B. Fernandez and B. Bussell. Bounds on the Number of Processors and Time for Multiprocessor Optimal Schedules. *IEEE Transactions on Computers*, C-22(8):745–751, Aug 1973.

[Fiedler and Baumgartl, 2004] Martin Fiedler and Robert Baumgartl. Implementation of a basic H.264/AVC decoder. Technical report, Technische Universität Chemnitz, June 2004.

[Freedman *et al.*, 2007] David Freedman, Robert Pisani, and Roger Purves. *Statistics*. W. W. Norton and Company, 4th edition, March 2007.

[Fujita and Nakagawa, 1999] Satoshi Fujita and Tadanori Nakagawa. Lower Bounding Techniques for the Multiprocessor Scheduling Problem with Communication Delay. In *Proc. of The International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 212–220, 1999.

[Fujita *et al.*, 2003] Satoshi Fujita, Masayuki Masukawa, and Shigeaki Tagashira. A Fast Branch-and-Bound Scheme for the Multiprocessor Scheduling Problem with Communication Time. In *ICPP Workshops*, pages 104–110, 2003.

[Gajski and Peir, 1985] Daniel D. Gajski and Jib-Kwon Peir. Essential Issues in Multiprocessor Systems. *IEEE Computer*, 18:9–27, June 1985.

[Garey and Johnson, 1978] Michael R. Garey and David S. Johnson. "Strong" NP-completeness results: motivation, examples and implications. *Journal of the Association for Computer Machinery (JACM)*, 25(3):499–508, July 1978.

[Gautama and van Gemund, 2000] H. Gautama and A. J. C. van Gemund. Static Performance prediction of data-dependent programs. In *2nd International Workshop on Software and Performance*, pages 216–226, Sep 2000.

[Gerasoulis and Yang, 1992] Apostolos Gerasoulis and Tao Yang. A Comparison of Clustering Heuristics for Scheduling DAGs onto Multiprocessors. *Journal on Parallel and Distributed Computing*, 16(4):276–291, Dec 1992.

[Gheorghita *et al.*, 2008] Stefan Valentin Gheorghita, Twan Basten, and Henk Corporaal. Scenario selection and prediction for DVS-aware scheduling of multimedia applications. *Signal Processing Systems*, 50(2):137–161, Feb 2008.

[Gil *et al.*, 2008] Luís Gil, Paulo Flores, and Luís Miguel Silveira. PMSat: a parallel version of MiniSAT. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 6:71–98, Sep 2008.

[Glasserman, 2004] Paul Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer, 2004.

[Goodwin and Wilken, 1996] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocations using 0–1 integer programming. *Journal on Software Practical Experiments*, 26(8):929–965, 1996.

[Govindarajan *et al.*, 2003] Ramaswamy Govindarajan, Hongbo Yang, José Nelson Amaral, Chihong Zhang, and Guang R. Gao. Minimum Register Instruction Sequencing to Reduce Register Spills in Out-of-Order Issue Superscalar Architectures. *IEEE Transactions on Computers*, 52(1):4–20, 2003.

[Graham *et al.*, 1979] Ronald L. Graham, Eugene L. Lawler, Jan K. Lenstra, and Alexander H.G. Rinnooy Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. In *Annals of Discrete Mathematics*, volume 5, pages 287–326. North-Holland, 1979.

[Gries and Keutzer, 2005] Matthias Gries and Kurt Keutzer. *Building ASIPs: The Mescal Methodology*. Springer, 2005.

[Gries, 2004] Matthias Gries. Methods for Evaluating and Covering the Design Space during Early Design Development. *Integration, the VLSI Journal*, 38(2):131–183, 2004.

[Gupta and Nadarajah, 2004] Arjun K. Gupta and Saralees Nadarajah, editors. *Handbook of Beta Distributions and its Applications (Statistics: a Series of Textbooks and Monographs)*. CRC Press, 1st edition, June 2004.

[Gupta, 2000] Pankaj Gupta. *Algorithms for Routing Lookups and Packet Classification*, chapter Minimum average and bounded worst-case routing lookup time on binary search trees. PhD thesis, Stanford University, 2000.

[Hall and Hochbaum, 1997] Leslie A. Hall and Dorit S. Hochbaum. *Approximation Algorithms for NP-Hard Problems*, chapter Approximation Algorithms for Scheduling. PWS Publishing Company, Boston, MA, 1997.

[Hamadi *et al.*, 2008] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySat: solver description. Technical Report MSR-TR-2008-83, Microsoft Research, May 2008.

[Hoang and Rabaey, 1993] Phu D. Hoang and Jan M. Rabaey. Scheduling of DSP Programs onto Multiprocessors for Maximum Throughput. *IEEE Transactions on Signal Processing*, 41(6):2225–2235, June 1993.

[Holliman *et al.*, 2003] Matthew J. Holliman, Eric Q. Li, and Yen-Kuang Chen. MPEG Decoding Workload Characteristics. In *Sixth Workshop on Computer Architecture Evaluation Using Commercial Workloads*, pages 23–34, 2003.

[Hooker and Ottosson, 1999] John N. Hooker and Gregor Ottosson. Logic-Based Benders Decomposition, Dec 1999. http://www.citeseer.ist.psu.edu/hooker95logicbased.html.

[Hooker and Yan, 1995] John N. Hooker and Hong Yan. Logic Circuit Verification by Benders Decomposition. In Vijay A. Saraswat and Pascal Van Hentenryck, editors, *Principles and Practice of Constraint Programming*, The Newport Papers, pages 267–288. MIT Press, Cambridge, MA, 1995.

[Horowitz *et al.*, 2003] Michael Horowitz, Anthony Joch, Faouzi Kossentini, and Antti Hallapuro. H.264/AVC baseline profile decoder complexity analysis. *IEEE Transactions on Circuits and Systems for Video Technologies*, 13(7):704–716, July 2003.

[Hsu *et al.*, 2005] Chia-Jui Hsu, Ming-Yung Ko, and Shuvra S. Bhattacharyya. Software Synthesis from the Dataflow Interchange Format. In *International Workshop on Software and Compilers for Embedded Processors*, Dallas, Texas, September 2005.

[Hu, 1961] Te C. Hu. Parallel Sequencing and Assembly Line Problems. *Operations Research*, 19(6):841–848, Nov 1961.

[Hubbard, 2007] Douglas W. Hubbard. *How to measure anything finding the value of 'intangibles' in business*. Wiley, 2007.

[Hughes *et al.*, 2001] Christopher J. Hughes, Sarita V. Adve, Rohit Jain, Jayanth Srinivasan, Praful Kaul, and Chanik Park. Variability in the Execution of Multimedia Applications and Implications for Architecture. In *ISCA '01: Proceedings of the 28th Annual Symposium on Computer Architecture*, pages 254–265, 2001.

[IBM Corp., 2007] IBM Corp. *Cell Broadband Engine Architecture*, 1.02 edition, October 2007.

[ILOG Inc.,] ILOG Inc. ILOG CPLEX Mathematical Programming (MP) Optimizer v10.1. http://www.ilog.com/products/cplex/.

[Intel Corp., 2002] Intel Corp. *Intel IXP2800 Network Processor Product Brief*, 2002.

[Iverson *et al.*, 1999] M.A. Iverson, F. Ozguner, and L.C. Potter. Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. *Proceedings of the Eighth Heterogeneous Computing Workshop, 1999 (HCW '99)*, pages 99–111, 1999.

[Jain and Grossmann, 2001] Vipul Jain and Ignacio E. Grossmann. Algorithms for Hybrid MILP/-CLP Models for a Class of Optimization Problems. *INFORMS Journal on Computing*, 13:258–276, 2001.

[Jin *et al.*, 2005] Yujia Jin, Nadathur Satish, Kaushik Ravindran, and Kurt Keutzer. An Automated Exploration Framework for FPGA-based Soft Multiprocessor Systems. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES'05)*, pages 273–278. ACM Press, 2005.

[Kasahara and Narita, 1984] Hironori Kasahara and Seinosuke Narita. Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing. *IEEE Transactions on Computers*, C-33(11), Nov 1984.

[Kerzner, 2003] Harold Kerzner. *Project Management: A Systems Approach to Planning, Scheduling and Controlling*. Wiley, 8th edition, 2003.

[Kim and Browne, 1988] S. J. Kim and J. C. Browne. A General Approach to Mapping of Parallel Computation upon Multiprocessor Architectures. In *Proceedings of the International Conference on Parallel Processing*, pages 1–8, August 1988.

[Kim and Han, 2005] Kyungjun Kim and Kijun Han. A lookup algorithm based on multiple tables for high-speed routers. *High Speed Networks, Vol.14, Iss.3*, pages 227–234, July 2005.

[Kim *et al.*, 2005] Sungchan Kim, Chaeseok Im, and Soonhoi Ha. Schedule-Aware Performance Estimation of Communication Architecture for Efficient Design Space Exploration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(5):539–552, May 2005.

[Kirkpatrick *et al.*, 1983] Scott Kirkpatrick, C. D. Gelatt Jr., and Mario P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):498–516, May 1983.

[Klastorin, 2003] Ted Klastorin. *Project Management: Tools and Trade-offs*. Wiley, 3rd edition, 2003.

[Koch, 1995] Peter Koch. Strategies for Realistic and Efficient Static Scheduling of Data Independent Algorithms onto Multiple Digital Signal Processors. Technical report, The DSP Research Group, Institute for Electronic Systems, Aalborg University, Aalborg, Denmark, December 1995.

[Kohler *et al.*, 2000] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

[Kwok and Ahmad, 1999a] Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and Comparison of the Task Graph Scheduling Algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, 1999.

[Kwok and Ahmad, 1999b] Yu-Kwong Kwok and Ishfaq Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[Lai and Xing, 2008] Tze Leung Lai and Haipeng Xing. *Statistical Models and Methods for Financial Markets*. Springer Publishers, 2008.

[Lawrence *et al.*, 1997] Steve Lawrence, Ah Chung Tsoi, and Andrew D. Back. Face recognition: A convolutional neural network approach. *IEEE Transactions on Neural Networks*, 8:98–113, 1997.

[Lee and Messerschmitt, 1987a] Edward A. Lee and David G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[Lee and Messerschmitt, 1987b] Edward Ashford Lee and David G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.

[Lenstra *et al.*, 1977] J.K. Lenstra, Alexander H.G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. In *Annals of Discrete Mathematics*, volume 1, pages 343–362. 1977.

[Liberatore *et al.*, 1999] Vincenzo Liberatore, Martin Farach-Colton, and Ulrich Kremer. Evaluation of Algorithms for Local Register Allocation. In *CC '99: Proceedings of the 8th International Conference on Compiler Construction*, pages 137–152, 1999.

[Mani and Orshansky, 2004] Murari Mani and Michael Orshansky. A New Statistical Optimization Algorithm for Gate Sizing. In *Proceedings of the 2004 IEEE International Conference on Computer-Aided Design (ICCAD)*, pages 272–277, 2004.

[Mani *et al.*, 2005] Murari Mani, Anirudh Devgan, and Michael Orshansky. An efficient algorithm for statistical minimization of total power under timing yield constraints. In *DAC 2005: Proceedings of the 42nd conference on Design Automation*, pages 309–314, 2005.

[Manolache *et al.*, 2004] Sorin Manolache, Petru Eles, and Zebo Peng. Schedulability Analysis of Multiprocessor Real-Time Applications with Stochastic Task Execution Times. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(4):706–735, Nov 2004.

[Manolache *et al.*, 2007] Sorin Manolache, Petru Eles, and Zebo Peng. *Real-Time Applications with Stochastic Task Execution Times: Analysis and Optimization*. Springer Netherlands, 2007.

[Masselos *et al.*, 2003] Konstantinos Masselos, Antti Pelkonen, Miroslav Cupak, and Spyros Blionas. Realization of wireless multimedia communication systems on reconfigurable platforms. *IEEE Journal of Systems Architecture: the EUROMICRO Journal*, 49(4-6):155–175, 2003.

[Moreira and Bekooij, 2007] Orlando M/ Moreira and Marco J. G. Bekooij. Self-Timed Scheduling Analysis for Real-Time Applications. *EURASIP Journal on Advances in Signal Processing*, 2007(Article ID 83710):1–14, 2007.

[Najm and Menezes, 2004] Farid N. Najm and Noel Menezes. Statistical timing analysis based on a timing yield model. In *Proceedings of the 41st Design Automation Conference*, pages 460–465, June 2004.

[National Instruments Inc.,] National Instruments Inc. NI LabVIEW. http://www.ni.com/labview.

[NVIDIA Corp., 2007] NVIDIA Corp. *NVIDIA CUDA Programming Guide*, November 2007. Version 1.1.

[Oh and Lee, 2003] Seunghyun Oh and Yangsun Lee. The Bitmap Trie for Fast IP Lookup. In *HSI 2003: Web and Communication Technologies and Internet-Related Social Issues*, pages 172–181, 2003.

[Olukotun and Hammond, 2005] Kunle Olukotun and Lance Hammond. The future of microprocessors. *ACM Queue*, 3(7):26–29, 2005.

[Orshansky and Keutzer, 2002] Michael Orshansky and Kurt Keutzer. A general probabilistic framework for worst-case timing analysis. In *Proceedings of the 39th Design Automation Conference*, pages 556–561, June 2002.

[Orsila *et al.*, 2006] Heikki Orsila, Tero Kangas, Erno Salminen, and Timo D. Hamalainen. Parameterizing Simulated Annealing for Distributing Task Graphs on Multiprocessor SoCs. In *Proc. of the International Symposium on System-On-Chip*, pages 1–4, November 2006.

[Orsila *et al.*, 2008] Heikki Orsila, Erno Salminen, and Timo D. Hämäläinen. *Simulated Annealing*, chapter Best Practices for Simulated Annealing in Multiprocessor Task Distribution Problems. I-Tech Education and Publishing KG, 2008.

[Papadimitriou and Ullman, 1987] Christos H. Papadimitriou and Jeffrey D. Ullman. A Communication-Time Tradeoff. *SIAM Journal of Computing*, 16(4):639–646, 1987.

[Papadimitriou and Yannakakis, 1988] Christos Papadimitriou and Mihalis Yannakakis. Towards an Architecture-Independent Analysis of Parallel Algorithms. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 510–513, New York, NY, USA, 1988. ACM Press.

[Poplavko *et al.*, 2003] P. Poplavko, T. Basten, M. Bekooij, J. van Meerbergen, and B. Mesman. Task-level Timing Models for Guaranteed Performance in Multiprocessor Networks-on-Chip. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 63–72, New York, NY, USA, 2003. ACM Press.

[Ravindran *et al.*, 2005] Kaushik Ravindran, Nadathur Satish, Yujia Jin, and Kurt Keutzer. An FPGA-based Soft Multiprocessor for IPv4 Packet Forwarding. In *15th International Conference on Field Programmable Logic and Applications (FPL-05)*, pages 487–492, Aug 2005.

[Rice, 1994] John Rice. *Mathematical Statistics and Data Analysis*. Duxbury Press, 2nd edition, June 1994.

[Rixner *et al.*, 1998] Scott Rixner, William J. Dally, Ujval J. Kapasi, Brucek Khailany, Abelardo López-Lagunas, Peter R. Mattson, and John D. Owens. A bandwidth-efficient architecture for media processing. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 3–13, 1998.

[Rudolph *et al.*, 1991] Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. A simple load balancing scheme for task allocation in parallel machines. In *SPAA '91: Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 237–245, 1991.

[Ruiz-Sánchez *et al.*, 2001] Miguel Ángel Ruiz-Sánchez, Ernst W. Biersack, and Walid Dabbous. Survey and Taxonomy of IP Address Lookup Algorithms. *Network, IEEE, Vol.15, Iss.2*, pages 8–23, March-April 2001.

[Ruppert, 2006] David Ruppert. *Statistics and Finance: An Introduction*. Springer Publishers, 1st edition, 2006.

[Satish *et al.*, 2007] Nadathur Satish, Kaushik Ravindran, and Kurt Keutzer. A Decomposition-based Constraint Optimization Approach for Statically Scheduling Task Graphs with Communication Delays to Multiprocessors. In *10th Conference of Design, Automation and Test in Europe (DATE-07)*, pages 57–62, New York, NY, USA, 2007. ACM Press.

[Sgall, 1997] J. Sgall. Online Scheduling - A Survey. In A. Fiat and G. Woeginger, editors, *On-Line Algorithms*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1997.

[Shah *et al.*, 2004] Niraj Shah, William Plishker, Kaushik Ravindran, and Kurt Keutzer. NP-Click: A Productive Software Development Approach for Network Processors. *IEEE Micro*, 24(5):45–54, Sep 2004.

[Shih *et al.*, 2004] Tse-Tsung Shih, Chia-Lin Yang, and Yi-Shin Tung. Workload Characterization of the H.264/AVC Decoder. In 5^{th} *IEEE Pacific-Rim Conference on Multimedia*, pages 957–966, 2004.

[Sih and Lee, 1993] Gilbert. C. Sih and Edward. A. Lee. A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, 1993.

[Sinha *et al.*, 2007] Debjit Sinha, Hai Zhou, and Narendra V. Shenoy. Advances in Computation of the Maximum of a Set of Gaussian Random Variables. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 26(8):1522–1533, Aug 2007.

[Slingerland and Smith, 2001] Nathan T. Slingerland and Alan Jay Smith. Cache Performance in Multimedia Applications. In *ICS 2001: Proceedings of the 15th International Conference on Supercomputing*, pages 204–217, June 2001.

[Srivastava *et al.*, 2004] Ashish Srivastava, Dennis Sylvester, and David Blaauw. Statistical optimization of leakage power considering process variations using dual-Vth and sizing. In *DAC 2004: Proceedings of the 41st conference on Design Automation*, pages 773–778, 2004.

[Sundaram *et al.*, 2009] Narayanan Sundaram, Anand Raghunathan, and Srimat T. Chakradhar. A framework for efficient and scalable execution of domain-specific templates on GPUs. In *To appear in IPDPS '09: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, 2009.

[Teich and Thiele, 1996] Jürgen Teich and Lothar Thiele. A Flow-Based Approach to Solving Resource-Constrained Scheduling Problems. Technical Report 17, Computer Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, April 1996.

[The MathWorks Inc., 2005] The MathWorks Inc. Simulink User's Guide, 2005. http://www.mathworks.com.

[Thiele, 1995] Lothar Thiele. Resource Constrained Scheduling of Uniform Algorithms. *VLSI Signal Processing*, 10(3):295–310, Aug 1995.

[Thies *et al.*, 2002] Willian Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *International Conference on Compiler Construction*, pages 179–196, Apr 2002.

[Tompkins, 2003] Mark F. Tompkins. Optimization Techniques for Task Allocation and Scheduling in Distributed Multi-Agent Operations. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, June 2003.

[van Dorp and Kotz, 2002] Johan René van Dorp and Samuel Kotz. A novel extension of the triangular distribution and its parameter estimation. *The Statistician*, 51(1):63–79, 2002.

[van Gemund, 1996] A. J. C. van Gemund. *Performance Modeling of Parallel Systems*. PhD thesis, Delft University of Technology, 1996.

[Veltman *et al.*, 1990] B. Veltman, B. J. Lagevreg, and J. K. Lenstra. Multiprocessor Scheduling with Communication Delays. *Parallel Computing*, 16(2-3):173–182, 1990.

[Verma *et al.*, 2004] Manish Verma, Lars Wehmeyer, and Peter Marwedel. Dynamic overlay of scratchpad memory for energy minimization. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 104–109, 2004.

[Visweswariah *et al.*, 2006] Chandu Visweswariah, Kaushik Ravindran, Kerim Kalafala, Steven G. Walker, Sambasivan Narayan, Daniel K. Beece, Jeff Piaget, Natesan Venkateswaran, and Jeffrey G. Hemmett. First-Order Incremental Block-Based Statistical Timing Analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 25(10):2170–2180, October 2006.

[Vose, 2008] David Vose. *Risk Analysis: A Quantitative Guide*. Wiley, 3rd edition, 2008.

[Wang *et al.*, 2006] Chao Wang, Aarti Gupta, and Malay Ganai. Predicate Learning and Selective Theory Deduction for a Difference Logic Solver. In *Proc. of the Design Automation Conference (DAC)*, pages 235–240, San Francisco, CA, USA, July 2006.

[Wang *et al.*, 2007] Gang Wang, Wenrui Gong, Brian DeRenzi, and Ryan Kastner. Ant Colony Optimizations for Resource and Timing Constrained Operation Scheduling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,*, 26(6):1010–1029, June 2007.

[Wawrzynek *et al.*, 2007] John Wawrzynek, David Patterson, M. Oskin, Shin-Lien Lu, Christos Kozyrakis, J.C. Hoe, D. Chiou, and Kriste Asanovic. RAMP: Research Accelerator for Multiple Processors. *IEEE Micro*, 27(2):46–57, March-April 2007.

[Xilinx Inc., 2004] Xilinx Inc. *Embedded Systems Tools Guide*. Xilinx Inc., Xilinx Embedded Development Kit, EDK version 6.3i edition, June 2004.

[Yang *et al.*, 1993] Jaehyung Yang, Ishfaq Ahmad, and Arif Ghafoor. Estimation of Execution times on Heterogeneous Supercomputer Architectures. In *ICPP '93: Proceedings of the 1993 International Conference on Parallel Processing*, pages 219–226, 1993.

[Yu *et al.*, 2005] Jia Yu, Jun Yang, Shaojie Chen, Yan Luo, and Laxmi Bhuyan. Enhancing Network Processor Simulation Speed with Statistical Input Sampling. In *HiPEAC 2005: International Conference on High Performance Embedded Architectures and Compilers*, pages 68–83, 2005.

[Ziou and Tabbone, 1998] Djamel Ziou and Salvatore Tabbone. Edge Detection Techniques - An Overview. *International Journal of Pattern Recognition and Image Analysis*, 8(4):537–559, 1998.